GI-Edition

Lecture Notes in Informatics

Stefan Wagner, Horst Lichter (Hrsg.)

Software Engineering 2013 Workshopband

(inkl. Doktorandensymposium)

26. Februar – 1. März 2013
Aachen

Proceedings

215

Stefan Wagner, Horst Lichter (Hrsg.):
Software Engineering 2013 – Workshopband

Stefan Wagner, Horst Lichter (Hrsg.)

Software Engineering 2013
–
Workshopband
(inkl. Doktorandensymposium)

Fachtagung des GI-Fachbereichs Softwaretechnik

26.02.–01.03.2013
in Aachen

Gesellschaft für Informatik e.V. (GI)

Lecture Notes in Informatics (LNI) - Proceedings
Series of the Gesellschaft für Informatik (GI)

Volume P-215

ISBN 978-3-88579-609-1
ISSN 1617-5468

Volume Editors
Stefan Wagner
Universität Stuttgart
Institut für Softwaretechnologie
Universitätsstr. 38
70569 Stuttgart
Email: stefan.wagner@informatik.uni-stuttgart.de
Horst Lichter
RWTH Aachen
Ahornstr. 55
52074 Aachen
Email: horst.lichter@swc.rwth-aachen.de

printed by Köllen Druck+Verlag GmbH, Bonn

Vorwort

Die Tagung *Software Engineering 2013* (SE 2013) fand vom 26.02. bis 01.03.2013 an der RWTH Aachen als die erste deutsche Multikonferenz zum Thema Software Engineering statt.

Diese integrierte neben der bekannten wissenschaftlichen Hauptkonferenz die industrieorientierte Konferenz *Software & Systems Engineering Essentials* (SEE 2013) und den seit vielen Jahren etablierten Workshop *Software Engineering im Unterricht der Hochschulen* (SEUH 2013). Damit entstand ein gemeinsames Forum für Software Engineering Professionals im deutschsprachigen Raum.

Ein wesentlicher Teil dieser Multikonferenz waren die am Vortag und parallel zur Konferenz stattfindenden Workshops und Tutorien. Sie boten für Interessierte die Gelegenheit, spezifische Themen im kleineren Kreis ausführlich und kontrovers zu diskutieren. Zusätzlich fand ein Doktorandensymposium statt, in dem der wissenschaftliche Nachwuchs Forschungsthemen im Bereich Software Engineering mit erfahrenen Hochschullehrern diskutieren konnte.

Dieser Workshopband enthält die Beiträge der Workshops

- *6. Arbeitstagung Programmiersprachen* (ATPS 2013),
- *First European Workshop on Mobile Engineering* (ME'13),
- *Traceability – Nutzung und Nutzen*,
- *Modellierung von Vorgehensmodellen – Paradigmen, Sprachen, Tools*,
- *3. Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systemen* (ENVISION2020),
- *Design For Future – Langlebige Softwaresysteme* (DFF) und
- *Zertifizierung und modellgetriebene Entwicklung sicherer Software* (ZeMoSS)

sowie alle im Doktorandensymposium präsentierten Beiträge

Unser Dank gilt den Organisatoren und Programmkomitees der einzelnen Veranstaltungen für ihre Arbeit und Engagement. Wir danken Rainer Koschke, Holger Schlingloff und Kurt Schneider für die Auswahl der Workshops und den Mitgliedern des Programmkomitees des Doktorandensymposiums – Uwe Assmann, Jürgen Ebert, Maritta Heisel, Manfred Nagl und Kurt Schneider – für die Unterstützung der jungen Promovierenden.

Ferner danken wir den vielen Helfern, die bei der Vorbereitung und Durchführung der Workshops mitgearbeitet haben, insbesondere Andreas Wortmann und Jan Oliver Ringert. Diesen Workshopband haben Ivan Bogicevic, Erica Janke, Jan-Peter Ostberg und Ana-Maria Dragomir erstellt. Vielen Dank dafür!

Stefan Wagner, Leitung Workshops und Tutorien
Horst Lichter, Leitung Doktorandensymposium

Stuttgart und Aachen, im Februar 2013

Inhaltsverzeichnis

ATPS 2013 – 6. Arbeitstagung Programmiersprachen

DFF – Design for Future 2013
Software Engineering für langlebige Systeme

ENVISION2020 – 3. Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme

ME'13 − First European Workshop on Mobile Engineering

Modellierung von Vorgehensmodellen − Paradigmen, Sprachen, Tools

Traceability – Nutzung und Nutzen

ZeMoSS – Zertifizierung und modellgetriebene Entwicklung sicherer Software

Doktorandensymposium

ATPS 2013 – 6. Arbeitstagung Programmiersprachen

Vorwort zur
6. Arbeitstagung Programmiersprachen (ATPS 2013)

Jens Knoop
TU Wien
knoop@complang.tuwien.ac.at

Janis Voigtländer
Universität Bonn
jv@informatik.uni-bonn.de

Die Arbeitstagung Programmiersprachen dient dem Austausch zwischen Forschern, Entwicklern und Anwendern, die sich mit Themen aus dem Bereich der Programmiersprachen beschäftigen. Für diese Tagung sind alle Programmierparadigmen und deren Sprachen von Interesse: imperative, objektorientierte, funktionale, logische, parallele, graphische Programmiersprachen, auch verteilte und nebenläufige Programmierung in Intra- und Internet-Anwendungen, sowie Konzepte zur Integration dieser Paradigmen. Die ersten vier Arbeitstagungen Programmiersprachen fanden 1997 in Aachen, 1999 in Paderborn, 2004 in Ulm und 2009 in Lübeck im Rahmen von GI-Jahrestagungen statt. Nach 2012 in Berlin findet die Tagung in diesem Jahr in Aachen zum zweiten Mal zusammen mit der GI-Tagung Software Engineering statt.

Auf den Aufruf zur Einreichung von Beiträgen sind 17 Arbeiten eingegangen. Nach ausführlicher Begutachtung und Diskussion hat das Programmkomitee zehn Beiträge zur Präsentation angenommen, die in diesem Tagungsband zusammengestellt und veröffentlicht sind. Thematisch decken die Beiträge einen breiten Querschnitt aktueller Forschungsthemen im programmiersprachlichen Bereich ab, insbesondere:

- Entwurf von Programmiersprachen und anwendungsspezifischen Sprachen
- Implementierungs- und Optimierungstechniken
- Analyse und Transformation von Programmen
- Typsysteme, Semantik und Spezifikationstechniken
- Modellierungssprachen, Objektorientierung
- Formale Methoden, Programm- und Implementierungsverifikation
- Werkzeuge und Programmierumgebungen
- Frameworks, Architekturen, generative Ansätze
- Erfahrungen bei exemplarischen Anwendungen

Im einzelnen wurden folgende eingereichte Beiträge angenommen:

- Till Berger und David Sabel:
  Parallelizing DPLL in Haskell
- Gergö Barany:
  Static and Dynamic Method Unboxing for Python
- Dennis Klassen:
  ViCE-UPSLA: A Visual High Level Language for Accurate Simulation of Interlocked Pipelined Processors
- Roland Lezuo, Gergö Barany und Andreas Krall:
  CASM: Implementing an Abstract State Machine based Programming Language
- Henning Heitkötter, Tim A. Majchrzak und Herbert Kuchen:
  MD^2-DSL — eine domänenspezifische Sprache zur Beschreibung und Generierung mobiler Anwendungen
- Steven Arzt, Kevin Falzon, Andreas Follner, Siegfried Rasthofer, Eric Bodden und Volker Stolz:
  How useful are existing monitoring languages for securing Android apps?
- Jürgen Graf, Martin Hecker und Martin Mohr:
  Using JOANA for Information Flow Control in Java Programs — A Practical Guide
- Reiner Jung, Christian Schneider und Wilhelm Hasselbring:
  Type Systems for Domain-specific Languages
- Jan Oliver Ringert, Bernhard Rumpe und Andreas Wortmann:
  From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems
- Baltasar Trancón y Widemann und Markus Lepper:
  Paisley: A Pattern Matching Library for Arbitrary Object Models

Über die eingereichten Beiträge hinaus freuen wir uns besonders, dass Jürgen Giesl unsere Einladung angenommen hat und einen Hauptvortrag zum Thema *Automated Termination Analysis: From Term Rewriting to Programming Languages* hält.

Abgerundet wird das Programm durch ein eingeladenes Impulsreferat von Wolf Zimmermann mit anschließender Podiums- und Publikumsdiskussion zum Thema *Modell-basierte Programmgenerierung und Methoden des Übersetzerbaus — Zwei Seiten derselben Medaille?*

Wir möchten uns bei allen Beteiligten für die Unterstützung zum Gelingen der ATPS 2013 bedanken. Zuallererst gilt unser Dank den Autoren und den eingeladenen Vortragenden für ihre Beiträge zum Programm. Wir danken außerdem den Programmkomiteemitgliedern und externen Gutachtern. Sie alle haben in kurzer Zeit engagiert und mit großem Einsatz daran gearbeitet, die Einreichungen zu begutachten und den Autoren fundierte

und nützliche Rückmeldungen zukommen zu lassen. Unser besonderer Dank gilt den Organisatoren der SE 2013 in Aachen für ihre Unterstützung, die ATPS 2013 zusammen mit der SE 2013 abzuhalten.

Wir hoffen, dass die ATPS 2013 für alle Teilnehmer ein interessantes und stimulierendes Forum ist und wertvolle Gelegenheiten zum Austausch von Ideen und zur Knüpfung und Vertiefung von Kontakten bietet.

Februar 2013

Jens Knoop
Janis Voigtländer

ATPS 2013 Tagungsorganisation

Tagungsleitung

Jens Knoop	TU Wien
Janis Voigtländer	Universität Bonn

Programmkomitee

Eric Bodden	TU Darmstadt
Sabine Glesner	TU Berlin
Clemens Grelck	Universiteit van Amsterdam
Michael Hanus	Universität Kiel
Matthias Hauswirth	Università della Svizzera italiana, Lugano
Christian Heinlein	Hochschule Aalen
Christoph Kessler	Linköpings Universitet
Raimund Kirner	University of Hertfordshire
Jens Knoop, Ko-Vorsitzender	TU Wien
Herbert Kuchen	Universität Münster
Michael Leuschel	Universität Düsseldorf
Rita Loogen	Universität Marburg
Christian W. Probst	Danmarks Tekniske Universitet, Lyngby
Volker Stolz	Universitetet i Oslo
Janis Voigtländer, Ko-Vorsitzender	Universität Bonn
Wolf Zimmermann	Universität Halle-Wittenberg

Externe Gutachter

Franziska Bathelt-Tok	TU Berlin
Steffen Ernsting	Universität Münster
Andreas Krall	TU Wien
Dan Li	University of Macao
Björn Peemöller	Universität Kiel
Marcel Pockrandt	TU Berlin
Ka I Pun	Universitetet i Oslo
Franz Puntigam	TU Wien
Fabian Reck	Universität Kiel
Markus Schordan	UAS Technikum Wien
David Schneider	Universität Düsseldorf
Daniel Stöhr	TU Berlin
Michael Zolda	University of Hertfordshire

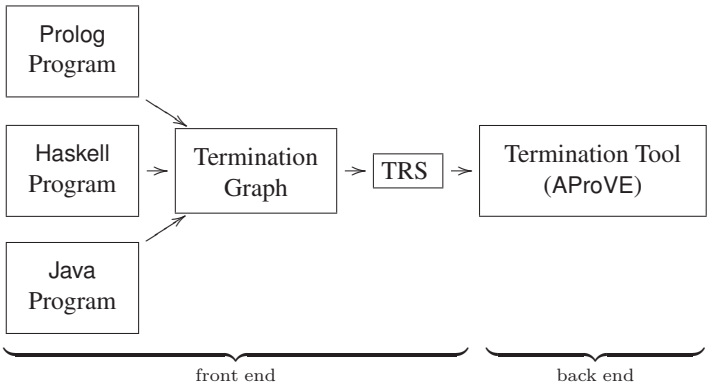
Automated Termination Analysis: From Term Rewriting to Programming Languages

Jürgen Giesl

LuFG Informatik 2
RWTH Aachen
Ahornstr. 55
52074 Aachen, Germany
giesl@informatik.rwth-aachen.de

Termination is a crucial property of programs. Therefore, techniques to analyze termination automatically are highly important for program verification. Traditionally, techniques for automated termination analysis were mainly studied for declarative programming paradigms such as logic programming and term rewriting. However, in the last years, several powerful techniques and tools have been developed which analyze the termination of programs in many programming languages including Java, C, Haskell, and Prolog.

In order to re-use the wealth of existing tools and techniques developed for termination analysis of term rewriting (see e.g., [GTSKF06, Zan03]), we developed a *transformational* methodology to prove termination of programs in different languages. In a *front end*, the program is automatically transformed into a term rewrite system (TRS) such that termination of the TRS implies termination of the original program. To obtain TRSs which are suitable for automated termination proofs, the front end proceeds in two steps. In the first step, the program is executed symbolically to generate a so-called *termination graph*. This graph represents all possible evaluations of the program in a finite way. In the second step, the edges of the graph are transformed to rewrite rules. Finally, existing rewriting techniques are used in the *back end* to prove termination of the resulting TRS.



We developed such approaches to prove termination of Prolog [GSSK+12, SKGS+10], Haskell [GRSK+11], and Java [BMOG12, BOG11, BOvG10, BSOG12, OBvG10], and integrated them into our termination tool AProVE [GST06]. As shown at the annual *International Termination Competition*,[1] AProVE is currently not only the most powerful tool for automated termination analysis of TRSs, but also for Prolog, Haskell, and Java. This shows that the proposed methodology for rewrite-based automated termination analysis indeed leads to competitive results.

References

[BMOG12] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated Termination Proofs for Java Programs with Cyclic Data. In *Proc. CAV '12*, LNCS 7358, pages 105–122, 2012.

[BOG11] M. Brockschmidt, C. Otto, and J. Giesl. Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting. In *Proc. RTA '11*, LIPIcs 10, pages 155–170, 2011.

[BOvG10] M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination Graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, LNCS 6463, pages 17–37, 2010.

[BSOG12] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated Detection of Non-Termination and `NullPointerExceptions` for Java Bytecode. In *Proc. FoVeOOS '11*, LNCS 7421, pages 123–141, 2012.

[GRSK+11] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *ACM TOPLAS*, 33(2):7:1–7:39, 2011.

[GSSK+12] J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic Evaluation Graphs and Term Rewriting – A General Methodology for Analyzing Logic Programs. In *Proc. PPDP '12*, pages 1–12. ACM Press, 2012.

[GST06] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.

[GTSKF06] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[OBvG10] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010.

[SKGS+10] P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs with Cut. *Theory and Practice of Logic Programming*, 10(4-6):365–381, 2010.

[Zan03] H. Zantema. Termination. In Terese, editor, *Term Rewriting Systems*, pages 181–259. Cambridge University Press, 2003.

[1] See `http://termination-portal.org/wiki/Termination_Competition`

Modell-basierte Programmgenerierung und Methoden des Übersetzerbaus — Zwei Seiten derselben Medaille?

Wolf Zimmermann

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
Von-Seckendorff-Platz 1
06120 Halle (Saale), Germany
wolf.zimmermann@informatik.uni-halle.de

In der vergangenen Dekade hat Modell-basierte Entwicklung in der Softwaretechnik zunehmende Bedeutung gewonnen, vgl. z.B. [TS07, SRC+12, JLM+12]. Aus Modellen, die in einer formalen Sprache definiert sind (Domänen-spezifische Sprache) wird Code generiert, der die Modelle implementiert. Da Domänen-spezifische Sprachen nicht selten starken Änderungen und Erweiterungen unterworfen sind, haben sich Werkzeugkästen wie beispielsweise das Eclipse Modeling Framework (kurz: EMF) etabliert [SBMP08], mit deren Hilfe die Codegeneratoren selbst generiert werden können. Zur Spezifikation Domänen-spezifischer Sprachen wird ein Metamodell definiert, aus dem dann mittels Modelltransformationen die Transformationsregeln in die Zielsprache spezifiziert werden. Neben dem Codegenerator für eine Domänen-spezifische Sprache werden beispielsweise mit der EMF-Technologie Editoren für die Domänen-spezifische Sprache generiert, die auch durch die Eclipse-Technologie in Programmier- und Anwendungsumgebungen eingebettet werden können.

Als grundlegende Technologie werden Metamodelle meist in graphischer Form (z.B. UML-Klassendiagrammen) oder durch eine kontextfreie Grammatik [EEK+12] definiert. Mittels OCL können Konsistenzbedingungen angegeben werden, die alle Modelle einer Domänen-spezifischen Sprache erfüllen müssen. Meist führen diese zu Laufzeitprüfungen. Oft werden zusätzlich im Code des Modell-basierten Codegenerators noch manuelle Ergänzungen und Änderungen vorgenommen. Gründe können Effizienzsteigerungen in der Codegenerierung oder weitere Überprüfungen sein. Eine Weiterentwicklung einer Domänen-spezifischen Sprache erfordert daher nicht selten eine grundsätzliche Revision und Überarbeitung des Codegenerators.

Die Aufgabenstellung für Übersetzer ist nahezu identisch: ein Programm in höherer Programmiersprache wird in ein Programm einer Maschinensprache oder einer anderen Hochsprache (Cross-Compiler) transformiert. In einem Übersetzer wird der Quelltext auf korrekte Syntax hin analysiert und in eine interne Datenstruktur, den abstrakten Syntaxbaum transformiert. Anschließend werden Konsistenzbedingungen wie beispielsweise Typkorrektheit analysiert. Im Falle eines Cross-Compilers wird der abstrakte Syntaxbaum in das Zielprogramm transformiert, ansonsten in eine Zwischensprache, die weiter in den

Maschinencode übersetzt wird. Hier haben sich z.T. seit Mitte der 1970er Jahre Werkzeuge [Joh75, KHZ82, DUP+82] und Werkzeugkästen zur Generierung von Übersetzern etabliert. Aktuelle Werkzeugkästen sind beispielsweise ANTLRv3 [Par07] oder Eli [KWS07]. Die Syntaxanalyse wird durch Angabe der Lexik als reguläre Ausdrücke, durch eine kontextfreie Grammatik für die konkrete Syntax und der Datenstruktur für die abstrakte Syntax angegeben. Die Konsistenzprüfungen können aus geordneten Attributierten Grammatiken generiert werden, die Transformation in die Zielsprache bzw. Zwischensprache wird aus Baumtransformationen generiert. Im generierten Übersetzer müssen keine Änderungen mehr vorgenommen werden, da Hilfsfunktionen Bestandteil der Spezifikationen sind. Hilfsfunktionen sind bereits in der Wirtssprache (der Sprache, in der der Übersetzer implementiert ist) definiert. Sie können deshalb problemlos eingebunden und geändert werden. Dadurch ist es nicht notwendig, den generierten Code anzufassen, und man kann agil einen Übersetzer Sprachkonzept um Sprachkonzept anreichern.

Als Fazit ergibt sich eine deutliche Korrespondenz zwischen der Generierung Modellbasierter Codegeneratoren und der Generierung von Übersetzern (vgl. auch [Jör11]): Die Begriffe Metamodell und Abstrakte Syntax sind konzeptuell identisch. Konsistenzbedingungen werden durch unterschiedliche Technologien wie OCL bzw. attributierte Grammatiken definiert und Modelltransformationen entsprechen konzeptuell Baumtransformationen. Für die praktische Entwicklung von Modell-basierten Codegeneratoren kann daher ohne Weiteres Übersetzerbautechnologie verwendet werden. Ein möglicher Vorteil durch die Verwendung von attributierten Grammatiken an Stelle von OCL wäre eine statische Prüfung der Konsistenz der Modelle anstatt dies auf Laufzeitprüfungen zu verlagern. Durch die Definition von Hilfsfunktionen in der Wirtssprache anstelle derer nachträglichen manuellen Integration in den Codegenerator wäre eine agilere und kostengünstigere Entwicklung von Modell-basierten Codegeneratoren möglich. Die für die industrielle Praxis wichtige Generierung der Editoren für Domänen-spezifische Sprachen und deren Einbettung in Programmier- und Anwendungsumgebungen müsste allerdings noch erfolgen, da dies durch Übersetzertechnologie bisher wenig Beachtung gefunden hat.

References

[DUP+82] S. Drossopoulou, J. Uhl, G. Persch, G. Goos, M. Dausmann, and G. Winterstein. An attribute grammar for Ada. *ACM SIGPLAN Notices*, 17(6):334, 1982.

[EEK+12] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: implementing domain-specific languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, pages 112–121. ACM, 2012.

[JLM+12] S. Jörges, A.L. Lamprecht, T. Margaria, I. Schaefer, and B. Steffen. A constraint-based variability modeling framework. *International Journal on Software Tools for Technology Transfer (STTT)*, 14:511–530, 2012.

[Joh75] S.C. Johnson. *Yacc: Yet another compiler-compiler*. Bell Laboratories, 1975.

[Jör11] S. Jörges. *Genesys: A Model-Driven and Service-Oriented Approach to the Construction and Evolution of Code Generators*. PhD thesis, Technical University of Dortmund, 2011.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A practical compiler generator*. Lecture Notes in Computer Science 141. Springer, 1982.

[KWS07] U. Kastens, W.M.C. Waite, and A.M. Sloane. *Generating Software from Specifications*. Jones & Bartlett Learning, 2007.

[Par07] T. Parr. *The definitive ANTLR reference: building domain-specific languages*. 2007.

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Addison-Wesley Professional, 2008.

[SRC+12] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer (STTT)*, 14:477–495, 2012.

[TS07] Sven Efftinge und Arno Hasse Thomas Stahl, Markus Völter. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 2007.

Parallelizing DPLL in Haskell

Till Berger and David Sabel

Computer Science Institute
Goethe-University Frankfurt am Main
till.berger@stud.uni-frankfurt.de
sabel@ki.informatik.uni-frankfurt.de

Abstract: We report on a case study of implementing parallel variants of the Davis-Putnam-Logemann-Loveland algorithm for solving the SAT problem of propositional formulas in the functional programming language Haskell. We explore several state of the art programming techniques for parallel and concurrent programming in Haskell and provide the corresponding implementations. Based on our experimental results, we compare several approaches and implementations.

1 Introduction

Due to the ongoing developments in hardware, multiprocessor and multicore programming is becoming more and more popular. To benefit from this development, on the one hand it is necessary to parallelize known algorithms by modifying the existing sequential algorithms, and on the other hand (in the best case easy to use) programming primitives for parallel and/or concurrent programming have to be developed and evaluated.

For the functional programming language Haskell [Mar10] several approaches for parallel and concurrent programming exist (for overviews see [PS09, Mar12]). Our motivation of this paper is to evaluate the possibilities for parallel and concurrent programming by implementing a typical use case for parallelization in several variants. As a result we compare the different implementations experimentally by measuring their performance.

As an easy to parallelize but interesting problem we chose SAT solving, i. e. the problem of answering the question whether or not a propositional formula is satisfiable. More concretely, as the existing sequential algorithm we used the Davis-Putnam-Loveland-Logemann (DPLL) procedure [DP60, DLL62] which decides satisfiability of propositional formula in clause normal form, by using unit propagation and case distinction.

There are several investigations of parallelizing this algorithm (see e. g. [BS96, ZBH96, HJS09]) which is an ongoing research topic. However, our goal is not to provide a very fast implementation of DPLL, but to compare, explain, and investigate several possibilities for parallelization in the functional programming language Haskell and how they can be adapted to this particular use case.

A similar approach has been undertaken in [RF09] where different strategies for paral-

lelizing a nondeterministic search in Haskell were implemented and analyzed. The DPLL procedure was used as an example application. Compared to their approach we also provide implementations using (implicit) futures, a "parallel and" (which behaves like an amb operator), and we use the Eval monad, which supersedes Haskells's older `par` combinator and appeared after the work of [RF09]. As a further difference, our performance tests include satisfiable as well as unsatisfiable formulas.

Outline of the paper In Section 2 we briefly describe the satisfiability problem and the (sequential) DPLL algorithm which solves this problem. We also will provide a simple implementation in Haskell. In Section 3 we present our implementations of parallel variants of DPLL, using several programming libraries available for parallel and concurrent programming in Haskell. We will also briefly explain the corresponding libraries. In Section 4 we present and interpret our experimental results. Finally, in Section 5 we conclude and list some open questions left for further research.

More details of the undertaken case study (including more test results and further motivation) can be found in [Ber12].

2 The DPLL Algorithm

The DPLL algorithm [DP60, DLL62] takes a propositional formula in conjunctive normal form (i. e. a set of clauses) as input and decides whether or not the formula is satisfiable. It uses unit propagation (i. e. the combination of unit resolution and subsumption) and case distinction as techniques. DPLL is the core of many modern SAT solvers like Chaff [zCh12, MMZ$^+$01] and zChaff [zCh12], GRASP [SS96] or MiniSat [Min12, ES03].

We introduce some notation. A propositional *atom* is a propositional variable (e. g. x). A *literal* is an atom x (called a *positive literal*) or a negated atom $\neg x$ (called a *negative literal*). We will use l, l_i for literals. With $\bar{l}$ we denote $\neg x$ if l is a positive literal x and x if l is a negative literal $\neg x$. A propositional formula is in *conjunctive normal form* iff it is of the form $(l_{1,1} \vee \ldots \vee l_{1,m_1}) \wedge \ldots \wedge (l_{n,1} \vee \ldots \vee l_{n,m_n})$. We use clause sets instead of conjunctive normal forms, i. e. we write $\{\{l_{1,1}, \ldots, l_{1,m_1}\}, \ldots \{l_{n,1}, \ldots, l_{n,m_n}\}\}$, where the set $\{l_{i,1}, \ldots, l_{i,m_i}\}$ is called a *clause*. Clauses of the form $\{l\}$ are called *unit-clauses*. For a clause set $\mathcal{C}$, a literal l is *pure* in $\mathcal{C}$ iff $\bar{l}$ does not occur in $\mathcal{C}$.

In its core the DPLL procedure can be described by Algorithm 2.1 shown in Figure 1. The shown DPLL algorithm only decides satisfiability, but it is easy to adapt it to also generate models for satisfiable formulas by setting the literal l of unit clauses $\{l\}$ used for unit propagation in line 3 to true and setting the pure literals l chosen in line 7 to true.

In practice, further improvements are used to speed up search by using so-called conflict-driven backjumping instead of backtracking and learning clauses (see e. g. [NOT06] for an overview). However, for our case study we wanted to keep the core algorithm simple, so we did not use these improvements. Hence, our implementation of the DPLL algorithm in Haskell (which additionally computes a model for satisfiable clause sets) is very close to

Algorithm 2.1.

Input: *A propositional clause set* $\mathcal{C}$
Output: `true` *(*$\mathcal{C}$ *is unsatisfiable) or* `false` *(*$\mathcal{C}$ *is satisfiable)*

$DPLL(\mathcal{C})$:
(1) **if** $\emptyset \in \mathcal{C}$ **then return** `true`;
(2) **if** $\mathcal{C}$ = *emptyset* **then return** `false`;
(3) **if** $\exists$ *unit-clause* $\{l\} \in \mathcal{C}$ **then**
(4) $\mathcal{C}_1$:= *remove all clauses in* $\mathcal{C}$ *that contain literal* l;
(5) $\mathcal{C}_2$:= *remove all literals* $\overline{l}$ *occuring in* $\mathcal{C}_1$;
(6) **return** $DPLL(\mathcal{C}_2)$;
(7) **if** $\exists$ *pure literal* l *in* $\mathcal{C}$ **then**
(8) $\mathcal{C}_1$:= *remove all clauses in* $\mathcal{C}$ *that contain literal* l;
(9) **return** $DPLL(\mathcal{C}_1)$;
(10) *Choose a variable* x *that occurs in* $\mathcal{C}$;
(11) **return** $DPLL(\mathcal{C} \cup \{\{x\}\}) \wedge DPLL(\mathcal{C} \cup \{\{\neg x\}\})$;

Figure 1: The DPLL procedure

the pseudo code of Algorithm 2.1. Leaving out some helper functions, the implementation in Haskell is shown in Fig. 2. The functions not shown here are `findUnit` which searches for unit clauses, `resolve` which performs unit propagation, and `findLiteral` to find a next decision literal used in the case distinction. Literals are represented by integers where negative numbers represent negative literals. Compared to the pseudo algorithm our implementation does not perform deletion of isolated literals since searching and deletion is very expensive, and the run time gets slower if deletion is included.

3 Parallelizing the DPLL Algorithm in Haskell

Due to its tree-like recursion (the case distinction in line 11 of Algorithm 2.1), the DPLL algorithm is obviously parallelizable by evaluating both recursive calls in parallel. This can be seen if we look at the execution graph of the sequential algorithm. Figure 3 shows one such graph for our Haskell implementation executed on the example clause set

$$\{\{p,r\},\{p,s\},\{p,\neg r,\neg s\},\{\neg p,q,r\},\{\neg p,q,s\},\\ \{\neg p,q,\neg r,\neg s\},\{\neg q,r\},\{\neg q,s\},\{\neg q,\neg r,\neg s\}\}\ .$$

A node shows the selected literal in each step where the root node is the first step. If a node has only one child node, then it is an execution of unit propagation. Whereas if it has two child nodes, then it is an execution of the last rule. In this case, the left edge represents the positive path (i. e. the literal was assigned `true`), and the right edge represents the negative path (i. e. the literal was assigned `false`). Leaf nodes have either of the values `true` or `false`. For the example all leaves are `true`, and thus the formula is unsatisfiable. If the tree contains a path ending in a leaf marked with `false`, then a (partial) model of

```
type Literal   = Int
type Clause    = [Literal]
type Model     = [Literal]
type ClauseSet = [Clause]

dpll :: ClauseSet → Model → Model
dpll [] model = model
dpll clauses model
 | [] ∈ clauses = []
 | otherwise =
    case findUnit clauses of
      Just u → dpll (resolve u clauses) (u:model)
      Nothing →
        let dlit         = findLiteral clauses
            positivePath = dpll (resolve dlit clauses) (dlit:model)
            negativePath = dpll (resolve (- dlit) clauses) ((- dlit):model)
        in case positivePath of
            [] → negativePath
            xs → xs
```

Figure 2: The DPLL algorithm in Haskell

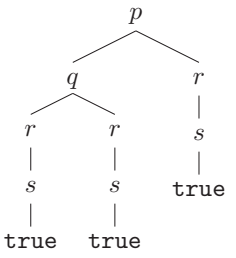


Figure 3: Example decision tree

the formula (i. e. a truth assignment) can be read off the corresponding path.

While sequential DPLL algorithms usually traverse the decision tree by a depth-first search, the idea for parallelization is to traverse all paths in parallel. Of course, this is a form of speculative parallelism, since perhaps unnecessary computations may be performed. On the other hand, even if we execute the parallel algorithm on a single processor (i. e. a concurrent evaluation which traverses the different paths in an interleaved manner), this may speed up the search by finding a model earlier.

Parallelization of the search with multiple processors is beneficial if all (or many) paths need to be searched (e. g. if the input clause set is unsatisfiable), or if a satisfying assignment is on a path that would be found late by the sequential search.

The parallel variant of the DPLL algorithm requires to synchronize the results of the com-

putations along different paths. For speeding up search, the algorithm should stop as early as possible. However, in the pure functional setting this requires to use a "parallel and" for synchronizing the results (i. e. an "and" operator which is non-strict in both of its arguments). Unfortunately, some libraries for parallel programming available for Haskell do not provide such a mechanism, and thus for several of our implementations we will use a "sequential and". However, in the impure world (using Haskell's `IO` monad), we will also provide an implementation that behaves like a "parallel and". As opposed to a "parallel and", a "sequential and" cannot achieve a speedup if the two paths traversed in parallel are both satisfiable, and one of them can be computed significantly faster than the other one.

Now, if we fork off computations at every decision point, the overhead for managing the parallel computations might get bigger than the speedup through parallelization if our clause sets get very small. Additionally, the number of processors (or cores) is limited by the hardware. So it makes no sense to parallelize every decision point. Instead, we restrict the number of created parallel computations.

Detecting the point for stopping parallel execution and going back to the sequential algorithm is not trivial as the formula size e. g. does not shrink predictably, and the detection should not be too expensive. Otherwise we may just as well have found a solution already with the computational power used by the detection.

For our implementations we have chosen a considerably simple approach: a global search depth that is used as an additional parameter for DPLL. This is almost without cost but makes us need to adjust for a particular formula size or group to be of general use.

3.1 Implementation in Haskell

The general model of our parallel implementation of DPLL in Haskell is shown in Figure 4. There are only few modifications w. r. t. the sequential implementation: An additional parameter for restricting the parallelization depth is inserted, and for the case distinction either the parallel execution is performed, or if the bound is exceeded, sequential computation is used. Of course, the code for the parallel execution is still missing and will be filled with several variants which we will explain in the subsequent sections. Since we also use Concurrent Haskell which is inside Haskell's `IO` monad we use a second variant, which is analogous to the program frame shown, but uses monadic `do` notation.

There are several methods for parallelizing code in Haskell. We have implemented some variants of the DPLL algorithm using the methods available within the Glasgow Haskell Compiler (GHC) and also using some others available by libraries. In particular, we used the Eval monad (on top of which evaluation strategies are built) [THLP98, MML+10], Concurrent Haskell [PGF96, Pey01, PS09], and its extension with futures [SSS11]. We also considered the Par monad [MNP11], which is available as a separate package (*monad-par*). We will discuss at the end of this section why we did not include an implementation using it in our tests.

In the following we are going for a mixed approach at explaining Haskell's parallelization capabilities and our respective implementations: We are discussing them at the same time

```
type Threshold = Int

dpllPar :: Threshold → ClauseSet → Model → Model
dpllPar _ []      model = model
dpllPar i clauses model
 | [] ∈ clauses = []
 | otherwise =
    case findUnit clauses of
     Just u → dpllPar i (resolve u clauses) (u:model)
     Nothing →
      let
       dlit         = findLiteral clauses
       positivePath = dpllPar (i-1) (resolve dlit clauses) (dlit:model)
       negativePath = dpllPar (i-1) (resolve (- dlit) clauses) ((- dlit):model)
      in if i > 0 then
           ... -- parallelization
         else case positivePath of
                [] → negativePath
                xs → xs
```

Figure 4: Program frame for all pure parallel implementations

and only explain the methods that are actually used. For a more thorough introduction into parallelism and concurrency within Haskell we refer to the tutorial [Mar12].

3.1.1 Implementation using the Eval Monad

The Eval monad [MML+10] is the successor of the older parallelization API with `par` and `pseq` [THLP98]. It delivers a more implicit approach to parallelization: The programmer annotates possible subexpressions for parallelization (by using `rpar`) and the runtime system *may* execute the evaluation in parallel. Concretely, the runtime system manages a pool of so-called *sparks*, and evaluation of `rpar` adds a new spark to the pool. If resources are available (i. e. a processor core is unused), the runtime system takes the next spark of the pool and executes it. More precisely, every HEC (Haskell Execution Context), which exists roughly for every processor core [MPS09], has its own spark pool where expressions given to `rpar` are stored. Whenever a processor core has no further work to do, it first looks at its own spark pool before stealing from another. The oldest sparks are taken out first, so overall, they are roughly executed in order of their creation.

`rpar` is used within the monad and execution is initiated with `runEval`. To wait for the evaluation of an expression, `rseq` is used. One important fact to note is that by default `rpar` and `rseq` only evaluate to weak head normal form (WHNF). Evaluation to normal form can be forced with `rdeepseq`, which can be combined with `rparWith` to evaluate a parallel computation completely. These functions are all basic evaluation strategies. Evaluation strategies are an abstraction layer on top of the Eval monad. They allow to separate the algorithm from its evaluation. The function `parList` e. g. represents a strategy that

evaluates all elements of a list in parallel. For more information on evaluation strategies we refer to [MML+10].

For the parallelization part of the DPLL algorithm we use the Eval monad as follows:

```
runEval $ do x ← rpar negativePath
             return (case positivePath of
                       [] → x
                       xs → xs)
```

The negative path is annotated to be evaluated in parallel, and the program continues by first evaluating the positive path. Note that we do not need to use the strategy `rdeepseq`, which ensures that the normal form of an expression is evaluated. If we inspect the DPLL algorithm more closely, we can see that `negativePath` is always evaluated to normal form by `rpar`, because at the point where we know if the result list is empty or not (i. e. if a model exists or not), the expression has already been fully evaluated.

This implementation is subsequently called `EvalT`. As sparks are more lightweight than threads because they are just references to the respective expressions, we also implemented a variant with unbound parallelization depth, which we call `Eval`.

3.1.2 Implementations using Concurrent Haskell

Concurrent Haskell [PGF96, Pey01, PS09] extends Haskell's `IO` monad with concurrent threads. These can be spawned by the primitive `forkIO`, and they can be killed by using `killThread`. Concurrent Haskell provides so-called `MVars` for synchronization of and communication between threads. An `MVar` is either empty or filled. The primitive `newMVar` creates a new (filled) `MVar`. The operation `putMVar` tries to fill an empty `MVar`. If the `MVar` is already filled, the calling thread is blocked until some other thread empties the `MVar`. Reading the content of an `MVar` is done with `takeMVar` which reads the content and empties the `MVar`. Similar to `putMVar`, it blocks the reading thread if the `MVar` is already empty and waits until it gets filled. Threads that are blocked on an otherwise inaccessible MVar are automatically garbage collected by GHC's garbage collector.

Using Concurrent Haskell, we implemented several variants. The first few use the above mentioned "sequential and" for parallelization, so the alternative path is only checked for a solution after the main one has finished computing. With these implementations we also compare the use of implicit vs. explicit futures as explained below. The "parallel and", where the result of the faster path is taken first, is implemented in one variant.

The program frame for the implementations using Concurrent Haskell differs slightly from Figure 4 because they return their result in the `IO` monad.

3.1.2.1 Concurrent Futures A *future* [BH77, Hal85] is a variable whose value is initially not known, but becomes available in the future. The value of a concurrent future is computed by a concurrent thread (in Haskell by a monadic computation in the `IO` monad, [SSS11]). Note that for computing the value of a *pure* expression, inside such a future, a monadic action which evaluates the expression must be created, for instance by using

the primitive `evaluate`. Like lazy evaluation, other threads using the future can continue evaluation until there is some data dependency forcing the value of the future. One distinguishes between explicit and implicit futures. Explicit futures require a `force` command to request the value of a future, while for implicit futures this request is performed automatically by the underlying runtime system.

Explicit futures are easy to implement using Concurrent Haskell: The future is represented by an `MVar`, and the concurrent thread writes its result into this `MVar`. The `force` command simply reads the content of the `MVar`. Hence, explicit futures can be implemented as follows, where we additionally return the `ThreadId` of the thread created:

```
type EFuture a = MVar a
efuture :: IO a → IO (ThreadId,EFuture a)
efuture act = do ack ← newEmptyMVar
                 tid ← forkIO (act >>= putMVar ack)
                 return (tid,ack)
force :: EFuture a → IO a
force x = readMVar x
```

Implicit futures are not implementable in Concurrent Haskell, but they can be implemented by using the primitive `unsafeInterleaveIO`[1], which delays the computation of a monadic action. The implementation in Haskell is:

```
future :: IO a → IO (ThreadId,a)
future act = do ack ← newEmptyMVar
                tid ← forkIO (act >>= putMVar ack)
                result ← unsafeInterleaveIO (takeMVar ack)
                return (tid,result)
```

Analogous to explicit futures, an `MVar` is used to store the result of the concurrent computation, but the code for creation of the future already contains the code for reading and returning the whole result. However, this second part of the code is delayed by `unsafeInterleaveIO`, which means that only if some other thread demands the value of the future, the code is executed.

Although this implementation makes use of the unsafe operation `unsafeInterleaveIO`, in [SSS12] it was shown that this specific use is safe since this extension of Haskell (i. e. Concurrent Haskell with implicit futures) is a conservative extension[2].

Our implementations for a concurrent DPLL algorithm with explicit and implicit futures are `ConE` and `Con` respectively. The parallelization in `ConE` is implemented as follows:

```
do (tid, npvar) ← efuture negativePath
   pp ← positivePath
   case pp of
       [] → force npvar >>= return
       xs → killThread tid ≫ return xs
```

[1] which is not part of the Haskell standard, but available in all major implementations of Haskell

[2] The result in [SSS12] does not include killing of the thread, but should be extensible to this situation

`Con`, using an implicit future, looks very similar, but the result does not need to be explicitly forced, it can directly be used:

```
do (tid, np) ← future negativePath
   pp ← positivePath
   case pp of
       [] → return np
       xs → killThread tid ≫ return xs
```

While implementing `Con`, we also implemented a variant where the order of the first two lines is switched, which resulted in the variant called `Con′`:

```
do pp ← positivePath
   (tid, np) ← future negativePath
   case pp of
       [] → return np
       xs → killThread tid ≫ return xs
```

Even though one might expect that this variant sequentializes the whole search, this is not true as our results will show. Through the use of the implicit future, the evaluation of the negative path is not forced with `return np`, but on the last branching point before, with `case pp of`. Thus, after nested recursive execution, the action `pp ←positivePath` is not strictly executed before the future for the negative path is created. In other words, as soon as the positive path is seen to have no result, a "pointer" to the negative-path computation is returned (`return np`). At the last branching point before, the negative path can now be forked off before the results of the positive path are completely forced with `case pp of`. In summary, the execution of the variant `Con′` is like first walking sequentially along the leftmost path of the decision tree, and then forking bottom-up so that the negative paths of different levels in the search tree are computed in parallel.

3.1.2.2 Checking the faster path first Using the mechanics of `MVars`, we can check the results of the paths in the order their computation finishes by letting them both write to the same `MVar`. The slower thread is blocked until the result of the faster one is read and taken out of the `MVar`. Then it can write to the `MVar`, and we can get its result with a second read on the `MVar`. If the first one already returned a solution, we kill both threads – one of the two `killThread` operations simply does nothing.

```
do pvar  ← newEmptyMVar
   tidp  ← forkIO (positivePath ≫= putMVar pvar)
   tidn  ← forkIO (negativePath ≫= putMVar pvar)
   first ← takeMVar pvar
   case first of
       [] → takeMVar pvar ≫= return
       xs → killThread tidp ≫ killThread tidn ≫ return xs
```

We call this implementation `Amb` (since it models McCarthy's *amb* [McC63]). Depending on which path (thread) finishes first, the resulting model (if one exists) may be different.

3.1.3 The Par Monad

As stated earlier, we did consider the Par monad, but excluded our implementation using it from the tests. The decision was made because the Par monad does not support speculative parallelism. It allows to create concurrent threads, and organizes communication between them with variables of the type `IVar`, which works similar to Concurrent Haskell's `MVar`. But there is no way to cancel an ongoing concurrent computation, and unreachable threads are not garbage collected like in Concurrent Haskell. An implementation using the Par monad would only be useful for (nearly) completely unsatisfiable formulas. There is a modification of the Par monad – found in the blog entry [Pet11] – that enables cancellation of threads, but we did not investigate an implementation using this modified Par monad.

4 Experimental Results

We tested several parallel implementations of the DPLL algorithm together with the sequential one using the Criterion package [Cri12]. The set of input formulas is part of the SATLIB project [HS00], which provides randomly generated 3-SAT formulas. The tested clause sets consist of 125 or 150 propositional variables and 538 or 645 clauses. The clause sets are divided into satisfiable and unsatisfiable sets (these are named to make the parameters explicit: the names are either *uuf-xxx-yyy* or *uf-xxx-yyy* where *uuf* means *unsatisfiable formula*, and *uf* means *satisfiable formula*; *xxx* is the number of variables, and *yyy* is the number of clauses). For the satisfiable formulas we tested 20 formulas of each set, and for the unsatisfiable formulas we tested 10 formulas of each set.

All tests were performed on a system with two quad-core processors of type AMD Opteron 2356 and 16 GB main memory. Every test was repeated 20 times to obtain the average execution time. The code was compiled with GHC version 7.4.2, and all variants apart from the sequential one were run with the parallel garbage collection switched on (which is the default); disabling the parallel garbage collection reduced performance for all parallel variants, only the sequential one ran faster without. Like the parallel variants, the sequential variant was compiled with threading support switched on but the runtime system was restricted to one core on execution.

The time we compare all runtimes to is the runtime of the sequential variant, which we call `Seq`. Subsequently, all runtimes are displayed as relative numbers; the runtime of the sequential variant is defined as 1.0 for every tested formula. Thus, for the sequential variant, we omit the relative numbers, but display the absolute runtimes for orientation.

The complete set of results – together with the source code – can be found at

`http://www-stud.cs.uni-frankfurt.de/~tilber/davis-putnam`.

Some of our results are shown in Figures 5 and 6. We also measured the space usage (by GHC's statistic output). Figure 7 shows some of these results.

Implementation	Threshold # Cores	1 2	2	8	2 4	8	3 8	12	24
uf125-538									
`Seq`	Mean	1.17 sec.							
`Amb`	Mean	**0.75**	1.49	4.47	1.05	2.43	1.03	1.78	1.99
	Median	0.95	1.37	1.77	0.86	0.97	**0.59**	0.69	0.74
`Con`	Mean	1.19	1.95	4.59	**1.14**	2.48	1.28	1.86	2.02
	Median	1.19	1.69	1.96	1.01	1.11	1.04	**0.71**	0.74
`Con′`	Mean	1.11	1.05	0.97	1.12	0.72	1.08	0.62	**0.56**
	Median	1.09	1.09	0.87	1.12	0.64	1.11	0.49	**0.44**
`EvalT`	Mean	1.02	0.98	1.07	0.92	0.87	0.91	**0.82**	0.83
	Median	1.08	1.07	1.07	1.02	0.90	0.88	0.58	**0.57**
`Eval`	Mean	1.01			0.87		0.82		
	Median	1.07			0.89		**0.61**		
uf150-645									
`Seq`	Mean	5.27 sec.							
`Amb`	Mean	**0.51**	2.49	10.33	1.31	5.57	1.28	4.62	5.77
	Median	0.41	0.78	1.59	0.43	0.86	**0.35**	0.64	0.77
`Con`	Mean	1.10	1.89	10.35	**1.03**	5.90	1.26	4.77	5.65
	Median	1.10	1.15	1.64	0.91	0.89	0.77	**0.65**	0.78
`Con′`	Mean	1.10	1.09	0.86	1.13	0.69	1.02	0.55	**0.48**
	Median	1.10	1.09	0.74	1.11	0.59	1.05	0.45	**0.37**
`EvalT`	Mean	1.00	1.08	0.96	0.89	0.78	0.95	0.74	**0.73**
	Median	1.07	1.03	0.98	0.80	0.69	0.68	0.48	**0.46**
`Eval`	Mean	0.94			0.78		**0.77**		
	Median	0.98			0.63		**0.47**		

Figure 5: Test results for satifisiable formulas (runtimes relative to `Seq`)

Implicit and Explicit Futures We did not include runtimes for the implementation with explicit futures (`ConE`) since they are almost the same as the runtimes for the implicit-future variant (`Con`). The reason is that the `force` command in the variant with explicit futures is almost exactly at the point where the result is needed. The only difference is that in the implicit-future variant, the negative path computation does not need to be evaluated before it is handed to the parent branching point. This difference is only essential for `Con′` where the negative path is forked off before the evaluation of the positive path is completely forced with `case pp of`.

Parallelization Depth for the Implementations using the Eval Monad Our results show that for using the Eval monad, the implementation with a depth bound (`EvalT`) performs not as good as the implementation without any bound (`Eval`). The numbers which are slightly higher for `Eval` can be attributed to measuring inaccuracy. In general,

Implementation	Threshold	1	2	8	2	8	3	12	24
	# Cores	2			4		8		
uuf125-538									
`Seq`	Mean	3.52 sec.							
`Amb`	Mean	0.75	0.70	0.57	0.59	0.33	0.47	0.21	**0.21**
	Median	0.72	0.69	0.57	0.59	0.33	0.45	0.21	**0.21**
`Con`	Mean	0.75	0.79	0.60	0.58	0.34	0.48	0.22	**0.22**
	Median	0.75	0.77	0.60	0.57	0.34	0.47	0.22	**0.22**
`Con′`	Mean	1.28	1.03	0.63	1.11	0.51	1.01	0.41	**0.36**
	Median	1.29	1.05	0.62	1.15	0.48	0.99	0.39	**0.34**
`EvalT`	Mean	0.68	0.66	0.55	0.51	0.31	0.41	0.19	**0.19**
	Median	0.66	0.61	0.54	0.50	0.31	0.41	0.19	**0.18**
`Eval`	Mean	0.54			0.30		**0.19**		
	Median	0.54			0.30		**0.18**		
uuf150-645									
`Seq`	Mean	10.80 sec.							
`Amb`	Mean	0.83	0.74	0.57	0.65	0.32	0.41	**0.20**	0.21
	Median	0.73	0.72	0.57	0.62	0.32	0.41	**0.20**	0.21
`Con`	Mean	0.87	0.82	0.59	0.69	0.33	0.43	**0.20**	0.21
	Median	0.79	0.78	0.59	0.70	0.33	0.42	**0.20**	0.21
`Con′`	Mean	1.31	0.98	0.60	1.07	0.44	0.97	0.32	**0.27**
	Median	1.34	0.91	0.59	0.98	0.44	0.96	0.31	**0.26**
`EvalT`	Mean	0.71	0.65	0.55	0.54	0.30	0.36	0.18	**0.17**
	Median	0.66	0.65	0.54	0.54	0.30	0.36	0.18	**0.17**
`Eval`	Mean	0.54			0.29		**0.18**		
	Median	0.54			0.29		**0.18**		

Figure 6: Test results for unsatisfiable formulas (runtimes relative to `Seq`)

the higher the threshold for `EvalT`, the shorter the runtime. Only in some cases a small threshold is faster, but overall, `Eval` has the best mean behavior. For unsatisfiable formulas the advantage of `Eval` is even clearer. A smaller partitioning of the search space seems to speed up the search more than the overhead for managing more sparks slows it down.

Comparison of `Eval` and `Con` As variants `Eval` and `Con` share the same parallelization approach – although the first uses parallelism, the latter concurrency –, one might expect them to perform similar. But `Con` is noticeably more fragile regarding the parallelization depth than `EvalT`. The reason is that the overhead of creating many threads in Concurrent Haskell is too high compared to the overhead that managing sparks in the Eval monad creates. Measuring the space behavior of both variants, we can see that `Con` consumes much more space than `Eval`, and consequently requires also more time for garbage collection. A relatively small threshold seems thus to yield the best results. But even considering that,

Implementation	Threshold	1	2	4	8	16	32	100
uf125-538								
`Seq`	Mean	4.01 MB						
`Amb`	Mean	1.2	1.5	2.6	4	6.9	7	7
`Con`	Mean	1.2	1.5	2.5	4.5	7.3	7.2	7.3
`Con′`	Mean	1	1.1	1.2	1.3	1.5	1.5	1.5
`EvalT`	Mean	1.2	1.6	1.6	1.6	1.6	1.6	1.6
`Eval`	Mean	1.6						
uf150-645								
`Seq`	Mean	4.77 MB						
`Amb`	Mean	1.2	1.6	3.4	7.8	17	17.8	18.4
`Con`	Mean	1.2	1.6	3.3	9.2	19	19	19.3
`Con′`	Mean	1	1.1	1.2	1.5	1.6	1.7	1.7
`EvalT`	Mean	1.3	1.7	1.7	1.7	1.7	1.7	1.7
`Eval`	Mean	1.7						

Figure 7: Space behavior for satisfiable formulas on four cores (relative to `Seq`)

`Eval` performs consistently better as it does not have such extreme outliers as `Con` (which result in the very high average runtimes for higher thresholds).

Parallelization Depth for `Con′` The implementation `Con′` also performs almost consistently better than `Con` for satisfiable formulas; but in another way than `Eval`. A noticeable difference is that for `Con′` the parallelization depth needs to be relatively high until measurable parallelization resulting in speedup happens. This is because of the parallelization being executed bottom-up. Like for `EvalT`, the higher the threshold, the faster the execution. Besides, a threshold of `100` is enough to completely parallelize the algorithm for all tested formulas – the maximum branching depth lies at about 50.

This also holds for unsatisfiable formulas. For these, `Con′` (with high threshold) is a bit slower than `Con` – but only on eight cores. This also indicates that parallelization happens later. As stated before, the positive path is evaluated only partly until a thread for the negative path is forked.

Inspecting the space behavior shows that `Con` consumes much more space than `Con′` for satisfiable formulas. As a consequence `Con` consumes more time for garbage collection than `Con′`. The reason might be that `Con′` creates a fewer number of threads, and the created threads are often necessary for the result of the computation.

Parallelization Depth for `Amb` The most ambiguous variant is `Amb`. It is even more fragile than `Con` regarding the parallelization depth. For satisfiable formulas a small depth is in most cases best so that the number of threads created equals the number of processor cores used. The mean is always worse for a higher threshold – meaning that there are

some extreme outliers that get worse the higher the threshold is. But as the median shows, a higher threshold yields a better runtime for some formulas. There are some extremely fast runtimes – only about 5 % of the sequential runtime – but that only happens on two cores with minimal threshold. Using more cores, the best single runtimes lie at about 18 %, and they are fewer. With a higher threshold, the time needed for garbage collection gets too high in most cases because of the high number of threads created. The increasing space usage can be seen in Figure 7.

For unsatisfiable formulas the almost reverse holds: Here, a threshold as high as possible is preferable. Though in some cases if it gets too high, the runtime slightly increases again. In this case, the thread-managing overhead seems to outweigh the better partitioning of the search space.

Comparing all Implementations Regarding unsatisfiable formulas, `Eval` performs best. But all parallel variants yield a noticeable advantage over the sequential one in this case. The relatively simple implementations perform very well when the whole decision tree needs to be traversed, and no speculative parallelism is actually performed.

For the satisfiable formulas, results are mixed. `Amb` is the fastest in some single cases where the advantage that it first checks the faster of both paths, translates to a very short runtime. But in many other cases the runtimes are extremely bad. This can be seen from the harsh difference between mean and median values for `Amb`. In these cases, the parallelization overhead seems to be too high. This is probably due to the fact that even more threads than with `Con` are created, which is also indicated by the fact that a high depth bound increases the runtimes even more than for `Con`. `Eval` is more stable but also has some outliers, which are less extreme. Its average results are thus looking better but, considering the number of cores utilized, not very well with only about 75 % of the sequential runtime for the medium sized formulas. In the average, `Con′` performs best for satisfiable formulas. Compared to the other two variants, runtimes are very stable resulting in mean values that are only slightly larger than the medians.

For `Eval`, the runtime variations were more noticeable with eight cores. Using less cores, the mean values are only a bit higher, whereas the median values improve with more cores. This suggests that the coordination costs get much higher in some cases when using more cores. Although the results for unsatisfiable formulas suggest that sparks behave indeed very cheap and that spark handling of the threaded GHC runtime system performs well, the results for satisfiable formulas show that there are cases where spark handling creates a noticeable overhead (though less much so than concurrent threads).

The depth bound approach is probably problematic if the search tree of a particular formula is very unbalanced. In that case the depth may in the worst case either be too big so that we do not stop parallelizing soon enough in the faster to solve path. This way, we lose time because of management overhead. Or it may be too low so that we lose parallelization potential in one path.

5 Conclusion

Concluding, our results show that the GHC runtime system performs very well when parallelizing sequential algorithms without speculative parallelism – in our case when unsatisfiable formulas are tested. Here, the most implicit approach, the Eval monad, yields the best results due to very little overhead. In case speculative parallelism is actually utilized when testing satisfiable formulas, our depth bound approach seems to be too simple though `Con′` performs surprisingly well compared to the other variants. For better results on satisfiable formulas, other techniques for bounding the parallelization depth – like trying to estimate the workload for a (partly reduced) formula – could be applied in further research. Furthermore, it might be interesting to see if a bottom-up forking like it happens with `Con′` is still advantageous in a more space-optimized implementation.

Acknowledgments We thank Manfred Schmidt-Schauß for reading this paper and for comments on this paper. We also thank the anonymous reviewers for their valuable comments.

References

[Ber12] T. Berger. Entwurf und Implementierung paralleler Varianten des Davis-Putnam-Algorithmus zum Erfüllbarkeitstest aussagenlogischer Formeln in der funktionalen Programmiersprache Haskell. Bachelorarbeit, Goethe-University Frankfurt am Main, Germany, 2012.

[BH77] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proc. Artif. intell. and prog. lang.*, pages 55–59. ACM, 1977.

[BS96] M. Böhm and E. Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. *Ann. Math. Artif. Intell.*, 17(3-4):381–400, 1996.

[Cri12] Criterion. Homepage, 2012. http://hackage.haskell.org/package/criterion.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[ES03] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. SAT 2003.*, volume 2919 of *Lecture Notes in Comput. Sci.*, pages 502–518. Springer, 2003.

[Hal85] R. H. Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, 1985.

[HJS09] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.

[HS00] H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In *SAT2000: Highlights of Satisfiability Research in the year 2000*, Frontiers in Artificial Intelligence and Applications, pages 283–292. Kluwer Academic, 2000.

[Mar10] S. Marlow. Haskell 2010 Language Report. `http://www.haskell.org/`, 2010.

[Mar12] S. Marlow. Parallel and Concurrent Programming in Haskell. In *CEFP 2011*, volume 7241 of *Lecture Notes in Comput. Sci.*, pages 339–401. Springer, 2012.

[McC63] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[Min12] MiniSat. Homepage, 2012. http://minisat.se/.

[MML+10] S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. W. Trinder. Seq no more: better strategies for parallel Haskell. In *Proc. Haskell 2010*, pages 91–102. ACM, 2010.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th DAC*, pages 530–535. ACM, 2001.

[MNP11] S. Marlow, R. Newton, and S. L. Peyton Jones. A monad for deterministic parallelism. In *Proc. Haskell 2011*, pages 71–82. ACM, 2011.

[MPS09] S. Marlow, S.L. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *Proc. ICFP 2009*, pages 65–78. ACM, 2009.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*). *J. ACM*, 53(6):937–977, 2006.

[Pet11] T. Petricek. Explicit speculative parallelism for Haskell's Par monad. http://tomasp.net/blog/speculative-par-monad.aspx, 2011.

[Pey01] S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS-Press, 2001.

[PGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23th POPL*, pages 295–308. ACM, 1996.

[PS09] S. Peyton Jones and S. Singh. A tutorial on parallel and concurrent programming in Haskell. In *6th AFP*, pages 267–305. Springer, 2009.

[RF09] F. Reck and S. Fischer. Towards a Parallel Search for Solutions of Non-deterministic Computations. In *Proc. ATPS 2009*, volume 154 of *LNI*, pages 2889–2900. GI, 2009.

[SS96] J. P. M. Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proc. ICCAD '96*, pages 220–227. IEEE Computer Society, 1996.

[SSS11] D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *13th PPDP*, pages 101–112. ACM, 2011.

[SSS12] D. Sabel and M. Schmidt-Schauß. Conservative Concurrency in Haskell. In *LICS*, pages 561–570. IEEE, 2012.

[THLP98] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithms + Strategy = Parallelism. *J. Funct. Program.*, 8(1):23–60, 1998.

[ZBH96] H. Zhang, M.P. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *J. Symb. Comput.*, 21(4):543–560, 1996.

[zCh12] zChaff. Homepage, 2012. http://www.princeton.edu/∼chaff/zchaff.html.

Static and Dynamic Method Unboxing for Python

Gergö Barany*
Institute of Computer Languages (E185)
Vienna University of Technology
Argentinierstraße 8
1040 Vienna, Austria
gergo@complang.tuwien.ac.at

Abstract: The Python programming language supports object-oriented programming using a simple and elegant model that treats member variables, methods, and various metadata as instances of a single kind of 'attribute'. While this allows a simple implementation of an interpreter that supports advanced metaprogramming features, it can inhibit the performance of certain very common special cases. This paper deals with the optimization of code that loads and then calls object methods.

We modify Python's compiler to emit special bytecode sequences for load/call pairs on object attributes to avoid unnecessary allocation of method objects. This can result in considerable speedups, but may cause slowdowns at call sites that refer to builtin functions or other special attributes rather than methods. We therefore extend this static compile-time approach by a dynamic runtime quickening scheme that falls back to the generic load/call sequence at such call sites.

The experimental evaluation of dynamic unboxing shows speedups of up to 8 % and rare slowdowns caused by as yet unresolved excessive instruction cache misses. A comparison with a common manual optimization of method calls in Python programs shows that our automatic method is not as powerful but more widely applicable.

1 Introduction

Dynamically typed interpreted programming languages like Python[1] are increasingly popular due to their ease of use and their flexibility that allows their application to a large range of problems. Such languages enable rapid prototyping and high programmer productivity because even large changes to a codebase are often local and do not involve changing many static type declarations.

Such flexibility comes at a price in program performance. A completely dynamic language cannot predict at compile time all the information necessary to specialize operations to certain data types. This dynamic behavior makes implementations using simple interpreters very attractive; just-in-time (JIT) compilation is also possible, but doing it right even given an existing JIT compiler for a different virtual instruction set involves considerably more effort than interpretation [CEI+12].

*This work was supported by the Austrian Science Fund (FWF) under contract no. P23303, *Spyculative*.

[1] `http://www.python.org/`

The main Python implementation is therefore an interpreter written in C and often referred to as CPython. It interprets a custom bytecode instruction set that is tightly coupled to Python's language semantics and uses an evaluation stack to pass data from one instruction to the next. In Python, everything is an object that can be assigned, inspected, and modified. This enables great expressiveness but can lead to inefficient behavior in common cases. In particular, methods are first-class objects as well, and method calls involve both a lookup in a dynamic data structure and the allocation of a method object on the heap which is then consumed and deallocated by the call.

Method calls are known to be slow in Python for these reasons. Previous work has proposed removing some of the complexity of the lookup by caching call targets [MKC+10] or call types [Bru10b]. We propose an orthogonal optimization: We do not modify lookup at all, but try to avoid the subsequent boxing step that allocates a method object on the heap.

The rest of this paper is organized as follows. In Section 2 we describe how Python looks up, boxes, and calls methods and similar objects and where inefficiencies lie. In Section 3 we propose a static compile-time solution to the problem of repeated boxing and unboxing of method objects and observe that it performs much worse than it should; Section 4 shows a solution to this problem by using quickening at runtime. Section 5 looks at two micro-benchmarks for an estimation of the best speedups possible using our method under maximally friendly and maximally adverse conditions. Section 6 discusses related work, and Section 7 concludes.

2 Method Calls in Python

As a prerequisite and motivation for our program transformation, we must first discuss which kinds of callable objects are available in Python, and how calls are implemented in the interpreters's bytecode. There are three major types of callable objects: Functions, methods, and 'builtin' operations. Functions behave as expected and are called with explicitly provided arguments. Methods are functions defined within class definitions and have a special first function argument `self` that refers to the call's receiver. Methods are called as expressions of the form `o.f(a, b)` where the `self` argument is bound to the value of `o` and the other arguments are passed as usual. A builtin is a piece of code implemented in C that may or may not have a `self` pointer and may or may not be called using the `o.f` attribute access syntax.

Given a variable (or more complex expression) `o`, a name `f`, and expressions `a` and `b`, the Python expression `o.f(a, b)` may mean one of several things:

- A call to a method named `f` associated with `o` or with `o`'s class or one of its superclasses, with the three arguments `o`, `a`, `b`, binding the value of `o` to the method's special `self` parameter.
- A call to a function implemented in Python and stored in `o`'s `f` field that is not a method, i. e., does not have a `self` pointer, and is called with only *two* arguments `a`

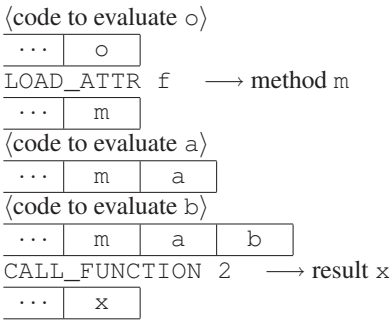


Figure 1: Abstract bytecode sequence and stack states for method call `o.f(a,b)`.

and `b`.

- A call to a 'builtin' function or method implemented in C and associated with `o`'s `f` field that may or may not have a `self` pointer and is called with two or three arguments, depending on its internal flags.
- A call to some other callable object (very rare).

Due to Python's dynamic nature which allows adding, deleting and modifying attributes of classes and any object instances at runtime, it is impossible to distinguish between these cases statically. The bytecode compiler therefore emits the same code sequence for each of these cases, and the interpreter must decide dynamically (i. e., at run time) how to execute each call based on the actual type of callable object encountered.

The bytecode sequence for this expression is illustrated in Figure 1. Code snippets are interleaved with an illustration of the state of the evaluation stack, which grows to the right. The basic call mechanism in Python is to push the function or method to call onto the stack, then push its arguments above it, and then to execute a call instruction to pop off all of these values and push the function's return value. The call instruction has a constant argument telling it how many arguments are on the stack above the function or method to call.

In the concrete case of method calls via an expression `o.f`, first the value of `o` is computed and placed on the stack, then a `LOAD_ATTR` instruction is executed to replace it with the value of its attribute `f`. Assume that `f` is defined as a method in the class of object `o` or some superclass of it. Method definitions in Python are simply function definitions located within class definitions and taking a special '`self`' pointer as their first argument. When a method is looked up, the lookup actually finds this function first and consults a 'descriptor' associated with it. This descriptor may be modified, but it will typically cause allocation of an actual *method* object containing both the function `f` and the receiver `o`. We call this operation *boxing* of the function, and refer to functions with a descriptor like this as *boxable* functions. Once the method object containing the function is allocated, it replaces the original object `o` on the stack.

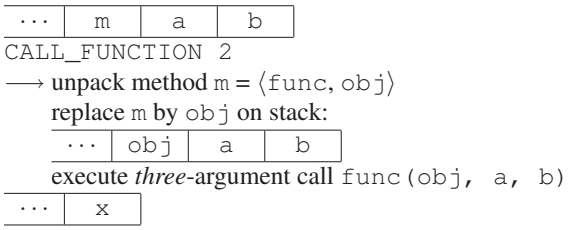


Figure 2: Unpacking a method object `m` to recover its `self` pointer.

Besides boxable functions, callable object attributes may also refer to plain Python functions that are returned by the lookup without boxing, or builtins, which are contained in a different type of object. Each of these also replace `o` on the stack.

Therefore, whatever the type of `o.f`, at this point `o` itself is no longer directly reachable from the stack. If the loaded value of `o.f` is a function without a `self` pointer, it will only be called with the two arguments `a` and `b` that are on the stack, so `o` is not necessary. However, if it is a method, the value of `o` must be recovered in order to pass it as the first of *three* arguments to the function call.

Figure 2 illustrates the Python interpreter's solution to this problem: When the object to be called is a method, its receiver is reachable from a pointer within it. The `CALL_FUNCTION n` instruction checks the type of the callable object below the arguments, at index $-\mathtt{n}-1$ from the stack top, and if it is a method, unboxes it. The receiver object replaces the method on the stack, and the function is executed with $\mathtt{n}+1$ arguments.

In summary, in the very common case of method calls of the form `o.f`, the Python interpreter performs a lookup for the code of `f` and boxes it up into a heap-allocated reference-counted method object $\langle \mathtt{o}, \mathtt{f} \rangle$ which is then almost immediately taken apart and deallocated by the corresponding call instruction. The method object cannot escape this bytecode sequence, its lifetime extends only from the attribute load to the function call. Thus, although the allocations are almost always served very quickly from a free list, all of these operations aren't really necessary, and we propose to replace them with an unboxing scheme.

3 Unboxed Method Calls

We propose to unbox method calls based on the observations that method calls are frequent in object-oriented Python programs (but not in numeric and some library-intensive benchmarks; we discuss this issue later), that boxing and unboxing is slow and redundant, and on the fact that a fast alternative exists: Once the method is unpacked and the `self` pointer is on the stack, the interpreter treats an n-argument method call identically to an $n+1$-argument call of a 'plain' Python function. Such calls are simple, so if we can set up the stack in the desired format right away, we can immediately execute the simple fast function call path.

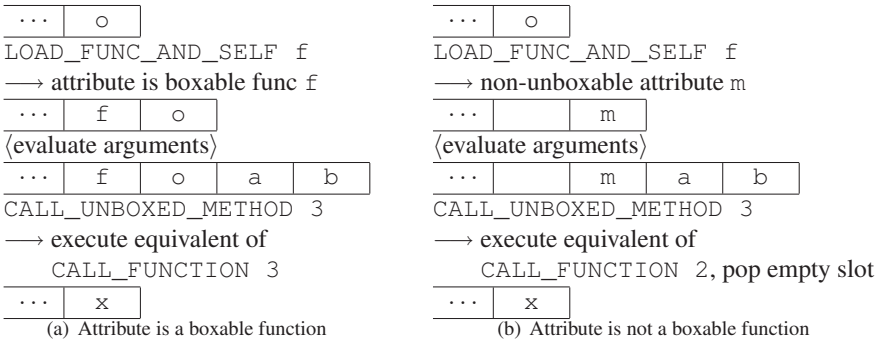


Figure 3: Unboxed attribute calls of the form `o.f(a,b)`.

3.1 Compile-Time Unboxing

The optimization is based on introducing two new bytecode instructions that specialize a `LOAD_ATTR`/`CALL_FUNCTION n` pair to unboxed method calls. During compilation from the program's abstract syntax tree (AST) to bytecode, we detect all cases where a call's target is an attribute access like `o.f`. In these cases, the modified compiler generates our new `LOAD_FUNC_AND_SELF` bytecode instead of `LOAD_ATTR`, and the new bytecode `CALL_UNBOXED n+1` instead of `CALL_FUNCTION n`. The semantics of these bytecodes is illustrated in Figure 3.

The `LOAD_FUNC_AND_SELF` bytecode is like `LOAD_ATTR` in that it performs a lookup of an attribute in the base object on the top of the stack. However, this bytecode is only used with `CALL_UNBOXED`, and its result takes up two stack slots. In the common case illustrated in Figure 3(a), the attribute lookup yields a function that `LOAD_ATTR` would box up with the receiver into a method object. We simply omit this boxing and place *both* the function and the receiver on the stack, with the function below the receiver. Function arguments are evaluated and pushed as usual, and when the corresponding `CALL_UNBOXED n+1` instruction is reached, the order of the function and its arguments on the stack is exactly as is needed for a plain function call, without any unboxing or on-stack replacement operations.

There is also the possibility, shown in Figure 3(b), that the attribute found by the lookup performed by `LOAD_FUNC_AND_SELF` is not a function that would be boxed up as a method. These are typically 'builtin' functions with an external implementation in C, or sometimes 'plain' Python functions that do not take a `self` argument. (These can be put into any slot of any object by simple assignment.) In these cases we do not need to put a receiver object onto the stack, but the following `CALL_UNBOXED n+1` needs to be able to distinguish between the unboxed method case and this non-method case. `LOAD_FUNC_AND_SELF` therefore leaves a stack slot empty (that is, we push a NULL pointer or other special sentinel value) and places the attribute it found above it. The corresponding call always checks this slot. If it is not empty, it must be a function, and

execution proceeds with a fast function call as in Figure 3(a); otherwise, a more elaborate call sequence based on the exact dynamic type of the looked-up attribute is initiated. After the call completes, the empty slot is removed from the stack before the call's result is pushed.

3.2 Experimental Evaluation

We implemented this method unboxing scheme in the Python 3.3.0 interpreter and evaluated it on a system with an Intel Xeon 3 GHz processor running Linux 2.6.30. Python was compiled with GCC 4.3.2 using the default settings, resulting in aggressive optimization with the `-O3` flag.

Despite the fact that Python 3 has existed for several years, its adoption by the community has been slow, which means that many popular benchmarks, such as the widely used Django web application framework, are only available in incompatible Python 2 versions. This restricts our choice of useful benchmark programs, but there is still a collection of useful though mostly quite small Python 3 compatible benchmarks available from `http://hg.python.org/benchmarks/`. We used these programs but excluded all those that took less than 0.1 seconds to execute a benchmark iteration because we found that these result in very inaccurate timings. We also excluded the micro-benchmarks that only test the speed of method calls (we look at these separately in Section 5) as well as the `2to3` benchmark that failed in some configurations even when using an unmodified Python interpreter.

In all of the experiments we used the fastest of five runs as the representative execution time of a benchmark.

Figure 4 shows the speedups we obtained by using static method unboxing versus the baseline, unmodified Python interpreter. We expected to observe speedups on object-oriented programs and unchanged performance on those benchmarks that do not use a large number of method calls. Instead, we were surprised to find that our proposed optimization tended to result in *slowdowns* on most programs. In a number of cases this is easy to explain as mis-speculation by the compiler that assumes that all attribute calls will go to unboxable methods implemented in Python. However, at least in the `fastpickle`, `fastunpickle`, `json_dump_v2`, and `json_load` benchmarks, this is simply not true: These benchmarks test the speed of libraries that are implemented as Python classes but with important operations implemented as callable attributes that are not Python methods but 'builtins' implemented in C. As we described above, we treat such calls by leaving the result of the lookup boxed, placing an empty slot on the stack below it, and checking that slot to decide how to perform the call. This overhead is not great, but it can result in a measurable slowdown if this slow path is hit in a benchmark that performs many of these calls. The following section proposes a solution for such cases of static mis-speculation.

Other slowdowns are more baffling: `nbody`, for instance, is a numeric benchmark that is aimed squarely at mathematical computation and contains no calls via attributes at all. No matter how we treat method calls, that treatment should not make an observable differ-

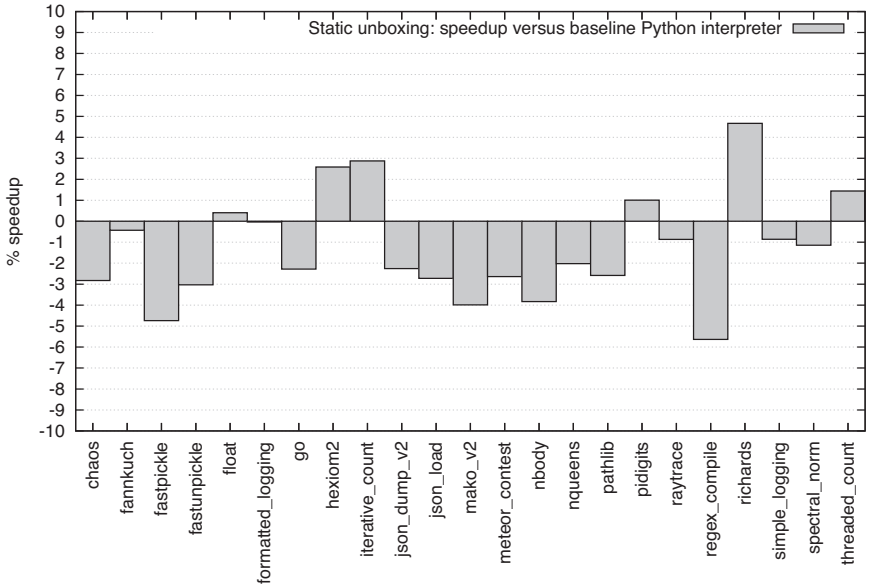


Figure 4: Performance impact of static method call unboxing at compile-time.

ence on such programs. We used the PAPI library[2] to collect information from hardware performance counters to better understand the behavior of the interpreter on such benchmarks. PAPI indeed verifies that the number of instructions executed on `nbody` for the baseline and our modified interpreter is essentially identical. However, the number of L1 instruction cache misses increases by a factor of 6–7. Our changes to the interpreter are small, but it appears that inserting the code for handling our two new bytecode instructions caused the compiler to make changes to the code layout that led to this very detrimental effect.

Overall, we observe that the benchmark suite contains quite few programs written in an object-oriented style and is thus inherently biased against optimizations aimed at Python method calls. Instruction cache misses may be difficult to fix because they depend on the code layout generated by the underlying compiler, but at least we must make an effort to correct the static mis-speculation at call sites that typically call builtins, not Python methods.

[2] `http://icl.cs.utk.edu/papi/`

4 Quickening to the Rescue!

The program transformation described in the previous section is completely static; the call sites where method unboxing is applied are determined by the bytecode compiler based only on the shape of the AST. However, we do not know at compile time which of these sites will actually call Python methods and which will call builtins, and as the experimental evaluation shows, this speculation on 'methods always' can fail and lead to diminished performance.

Since purely static speculation does not seem like a good approach, we therefore move on to a more dynamic approach using quickening. Quickening [LY96, Bru10a] is an operation that modifies code at runtime, typically after some sort of initialization has taken place or where data collected during previous execution of an instruction suggests that another variant of the instruction may execute more efficiently. Quickening can be applied to specialize code to particular data types; this often pays off because even in highly dynamic languages like Python, many pieces of code are 'dynamically static' in their usage of types. That is, even if operand types for given instructions cannot be predicted beforehand, they often tend to remain relatively constant over single program runs [DS84].

4.1 Quickening of Non-Unboxed Calls

In our dynamic method unboxing implementation, we specialize every call site separately depending on the type of the attribute loaded to serve as a call target. If a lookup/call pair referred to an unboxable method in the past, we expect it to also do so in the future; if, however, the attribute was a builtin or other non-method function, we assume that that call site will tend to call such non-methods in the future.

We therefore modified our implementation of the `LOAD_FUNC_AND_SELF` bytecode to use quickening in cases where the attribute it finds is not a boxable function. In such cases, we immediately fall back to the behavior of the baseline interpreter. Instead of using the path shown in Figure 3(b) with an empty stack slot, in this case `LOAD_FUNC_AND_SELF` simply pushes the attribute to the stack and immediately quickens itself to (i. e., replaces itself by) `LOAD_ATTR`. As the call target is now boxed, the following call can no longer be `CALL_UNBOXED`, so we immediately quicken it to `CALL_FUNCTION`. That is, at this call site this and all future calls will be treated exactly as in the regular Python interpreter.

To quicken the call corresponding to a given attribute lookup, we need to locate that call; this is implemented simply as a scan forward in the instruction stream over the argument evaluation code to the call. Computation of the arguments may also involve lookup/call pairs, but these are always strictly nested, so there is no ambiguity as to which lookup belongs to which call. A linear scan over the instructions may seem expensive, but the call is typically very close to the lookup (never more than 200 bytes later in our benchmarks, and typically much closer), and quickening is rare, so the total overhead is acceptable.

An alternative implementation without a forward sweep over the code would be to place a special marker on the stack as a signal to the call instruction to quicken itself. This

does not work in practice, however: Exceptions may (and do!) occur during argument evaluation, so the call instruction may not be reached. The next time the lookup/call pair would be executed, the lookup would already be quickened, but the call would not. As they have different expectations about the organization of the stack, this would lead to program failures. We must therefore quicken both the lookup and the call at the same time. A possibly slightly more efficient variant would be to associate an offset with the lookup that specifies exactly the distance to the corresponding call. For simplicity, we have not implemented this more complex variant yet.

If the attribute found by `LOAD_FUNC_AND_SELF` is a boxable method, we perform an unboxed method call as before, and no quickening is needed.

Figure 5 shows a graph of all the states that an attribute call site can be in, and the transitions between the states. In the quickening variant we introduced yet another pair of lookup/call instructions, called `LOAD_CALLABLE_ATTR` and `PSEUDO_CALL_ATTR`, respectively. The idea is to generate only these instructions in the compiler. The first time an occurrence of `LOAD_CALLABLE_ATTR` is executed, it quickens itself to either `LOAD_FUNC_AND_SELF` or to `LOAD_ATTR`, depending on whether the attribute it found is a method or not. If it is a method, it is unboxed on the fly, and the instruction is quickened to `LOAD_FUNC_AND_SELF`.

`PSEUDO_CALL_ATTR` is never executed and does not even have an implementation. As it is always preceded by `LOAD_CALLABLE_ATTR`, it will always be quickened to either `CALL_UNBOXED` or `CALL_FUNCTION` before it is reached.

Note the direction of transitions in the graph: Once a call site is quickened to the baseline case of `LOAD_ATTR`/ `CALL_FUNCTION`, it can never move back to unboxing, even if many of the future calls might profit from it. During testing, we found that such cases appear to be very rare or nonexistent in real Python programs, although it is easy to write simple counterexamples. In any case, `LOAD_ATTR`/ `CALL_FUNCTION` is a good, heavily optimized baseline implementation to fall back to.

4.2 Experimental Evaluation

Figure 6 shows the result of evaluating unboxing with quickening on the same benchmark set and same experimental setup as in Figure 4; the scales are identical to enable a quick visual comparison. We note the general trend that most results are much better with quickening than with the purely static compile-time method. In particular, the number of benchmarks exhibiting slowdowns has been reduced, and where slowdowns remain, they are less pronounced than before. Several object oriented benchmarks that showed speedups before are now even faster. The most impressive case is `richards`, where even more attribute calls are now handled as efficiently as possible.

We again investigated the case of the large 3 % slowdown on `regex_compile` in some detail. Using PAPI, we found that while the number of instructions executed was reduced by about 0.5 % due to unboxed method calls, we still incur about 20 % more instruction cache misses than the baseline. Again, we believe that this is due to the larger and more

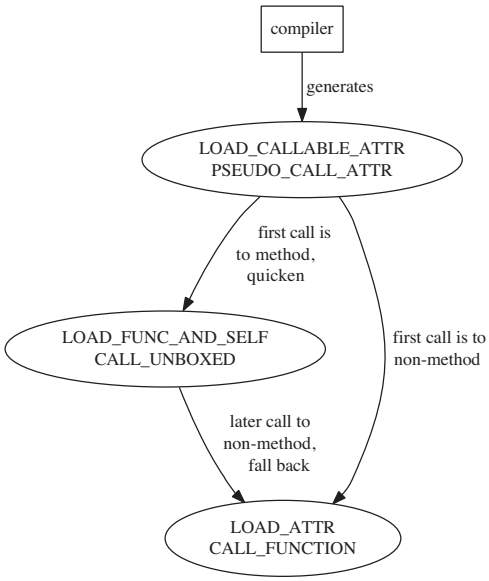


Figure 5: Possible states of bytecode pairs at each attribute call site.

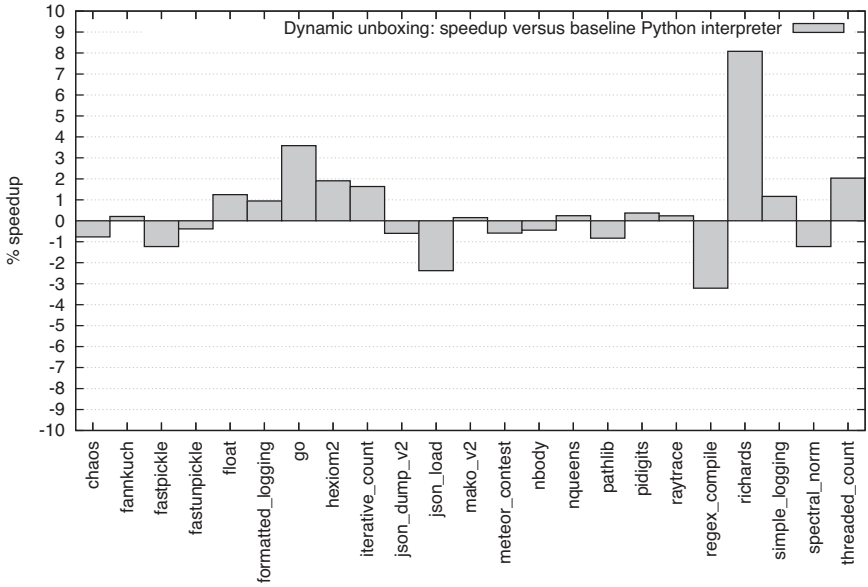


Figure 6: Performance impact of dynamic method call unboxing by quickening.

complex code in the interpreter loop due to our bytecode instruction set extensions. In fact, we also observe this slowdown and the cache misses if we leave our bytecodes in place but use an unmodified compiler that never generates these instructions. That is, adding these instructions incurs a cost even if they are never executed, simply due to code size and layout.

During initial testing of the quickening implementation, we also observed unexpectedly bad results for a number of benchmarks that should be able to take good advantage of unboxing. In particular, the `hexiom2` program showed a 2 % slowdown, which is strange given that the non-quickened version performs well. We suspected that this might be due to type checks in our implementation of `CALL_FUNC_AND_SELF`, which must always check whether the attribute to return is a boxable function. In the hope of optimizing code layout, we tried using GCC's `__builtin_expect` mechanism to annotate two `if` statements with the information that we typically expected their conditions to succeed. These annotations were indeed what was needed to turn a 2 % slowdown into a 2 % speedup.

These observations suggest that the control flow of a typical bytecode interpreter—a loop containing a large `switch` statement containing computed gotos to other cases—may be confusing enough to compilers to make such manual annotations worthwhile. It might also be useful to try to minimize code size within the interpreter loop as much as possible. In particular, researchers modifying such bytecode interpreters should be aware that adding new bytecode instructions can come with an intrinsic cost even if the modified instruction set helps perform 'less work' in terms of the number of executed instructions.

Implementation size. Overall, our changes to the Python interpreter to accommodate both static and dynamic method unboxing comprise adding 569 lines of code including extensive comments and commented-out debug code. The implementations are cleanly separated by preprocessor directives that must be specified at compile time. This way, we can build the baseline, static, and dynamic unboxing versions of the interpreter from the same codebase, and the purely static implementation does not have the code size overhead (and resulting possible additional instruction cache misses) of the dynamic version.

5 Method Unboxing vs. Method Caching

We excluded call-oriented micro-benchmarks from the experimental evaluation in the previous sections because the speedups we achieve here do not translate to speedups in real applications, which perform other work besides simple function or method calls. However, we perform two experiments here to obtain a rough comparison with optimizations based on caching of method call targets.

The `call_method` benchmark performs a large number of method calls via method attributes; Figure 7(a) shows an illustrative code snippet from the benchmark. All the call sites in this program are monomorphic, so aggressive optimizations are possible in theory: All calls could be resolved before runtime, or at least cached at each call site. We do not

```
class Foo(object):
   def foo(self, a, b, c, d):

     # 20 calls
     self.bar(a, b, c)
     self.bar(a, b, c)
     ...

def test_calls(n):
   f = Foo()

   for _ in xrange(n):
     # 20 calls
     f.foo(1, 2, 3, 4)
     f.foo(1, 2, 3, 4)
     ...
```

(a) Original benchmark

```
class Foo(object):
   def foo(self, a, b, c, d):
     self_bar = self.bar
     # 20 calls
     self_bar(a, b, c)
     self_bar(a, b, c)
     ...

def test_calls(n):
   f = Foo()
   f_foo = f.foo
   for _ in xrange(n):
     # 20 calls
     f_foo(1, 2, 3, 4)
     f_foo(1, 2, 3, 4)
     ...
```

(b) Manual caching of methods

Figure 7: The `call_method` micro-benchmark and a common manual optimization, caching methods in local variables.

perform any caching but are able to unbox all call sites, saving the method allocation and deallocation overhead associated with every call. On this micro-benchmark we achieve a speedup of 13 %, which we therefore assume is the best speedup that can ever be achieved using our method unboxing scheme (until we manage to fix instruction cache issues).

To put this into perspective, we compared our result to an optimization that is folklore in the Python community and applied widely in the Python library sources and other projects: Repeated calls (such as in a loop) of the same method on the same receiver object can be optimized by assigning the method object to a local variable and performing the calls through that variable. Figure 7(b) illustrates this transformation. The optimization effect here is due to avoiding repeated lookups of the same value and allocating/deallocating a method object at each call site. At the place of the call, the method object need only be fetched from the local variable, which is fast, and unboxed for the call, which is also fast since the object need not be garbage collected because a reference to it remains in the local variable. This manually optimized program achieves a speedup of 39 % over the original; as this is is considerably larger than our unboxing speedup, we conclude that there is still some room for better caching of method calls in Python.

As a second micro-benchmark, we consider `call_method_unknown`, which involves four classes containing identically-named methods that call each other. However, this benchmark is designed to foil any attempt at caching of receivers or their types, as its documentation explains: "To make the type of the invocant unpredictable, the arguments are rotated for each method call. On the first run of a method, all the call sites will use one type, and on the next, they will all be different."[3] Since the types will be different at

[3]The code is too convoluted to be excerpted here; see `http://hg.python.org/benchmarks/file/43f8a0f5edd3/performance/bm_call_method_unknown.py`

different times, a code transformation as in Figure 7 is not applicable, and any automatic caching would presumably incur many cache misses. Our unboxing approach, however, achieves a speedup of 12 % on this micro-benchmark. This is similar to the speedup on the simple method call benchmark and is to be expected, since in our scheme lookups are always performed as usual, so changing targets should not impact the interpreter's performance at all.

6 Related Work

An optimization that is very similar to our static approach was discussed in the Python community in 2003[4]. This change was never accepted into the Python distribution, apparently due to slowdowns that were similar to the ones we observe due to instruction cache misses.

Other than this, the most directly relevant work by Mostafa et al. [MKC+10] investigates various optimizations for the Python interpreter, including caching of attribute loads and loads from global variables. Using different variants of a caching scheme for `LOAD_ATTR` they achieved speedups of about 3–4 % on average, more if caching of global lookups was also enabled. Note that we do not report an average speedup for our transformations because our benchmark suite is biased against optimizations targeted at method calls, while the benchmarks used by Mostafa et al. appear to be mostly object-oriented, and optimized attribute lookups benefit most of their programs.

It is not clear how these speedups relate to ours because they used Python 2.6 and do not specify the minor version number; at some point during the 2.6 series, the official Python interpreter got some caching of attribute lookups built in, and we cannot tell whether their caching implementation came before or after this change. In any case, our results in comparison to Python 3.3 which definitely uses caching show that unboxing (which Mostafa et al. do not perform) is orthogonal to caching and can be combined with it profitably.

We use the term quickening adapted from Brunthaler [Bru10b, Bru10a] to refer to the operation of replacing an instruction with a different version, although the operation itself appears to be folklore. 'Quick' instructions appeared at the latest in early Java VMs [LY96]. A similar optimization is inline caching, which replaces not opcodes but instruction arguments in the instruction stream [DS84]. A similar optimization is also possible in interpreters that work not on bytecode but directly on abstract syntax trees [WWS+12].

The performance impact of code layout in the interpreter loop was recently investigated by Brunthaler [Bru11] and McCandless and Gregg [MG11], who report speedups of up to 14 % and 40 %, respectively.

[4] `http://mail.python.org/pipermail/python-dev/2003-February/033410.html`, `http://bugs.python.org/issue709744`

7 Conclusions and Future Work

We presented a simple compile-time approach to improve the performance of Python method calls by unboxing the method object consisting of a receiver and a function pointer. This approach works but is limited by Python's dynamic nature and the fact that things that appear syntactically as method calls might be calls to other kinds of objects with different calling conventions. We therefore introduce a dynamic improvement that rewrites the code at execution time depending on the actually observed kind of call.

This paper is the first step in a project aimed at a detailed understanding of Python and other high-level language interpreters. Instrumenting the interpreter to collect statistics about the frequency and execution time of bytecode instructions, we noticed a correlation between occurrences of `LOAD_ATTR` and `CALL_FUNCTION`, which are both frequent and expensive, and designed an optimization to remove some of their shared overhead.

The next step is a larger-scale study of Python 3 programs and their behavior during interpretation. We intend to generalize the results of previous investigations that focused on studies of reflective and other dynamic behavior in Python programs [HH09] and on optimizing certain classes of instructions based on such quantitative data [MKC+10].

Acknowledgements

The author would like to thank Stefan Brunthaler and the anonymous reviewers for suggestions that helped to improve the paper.

References

[Bru10a] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 1–14, New York, NY, USA, 2010. ACM.

[Bru10b] Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 429–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[Bru11] Stefan Brunthaler. Interpreter instruction scheduling. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 164–178, Berlin, Heidelberg, 2011. Springer-Verlag.

[CEI+12] Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 195–212, New York, NY, USA, 2012. ACM.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[HH09] Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In Bernard Mans, editor, *ACSC*, volume 91 of *CRPIT*, pages 17–25. Australian Computer Society, 2009.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Java Series. Addison-Wesley Longman, 1st edition, 1996.

[MG11] Jason McCandless and David Gregg. Optimizing interpreters by tuning opcode orderings on virtual machines for modern architectures: or: how I learned to stop worrying and love hill climbing. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 161–170, New York, NY, USA, 2011. ACM.

[MKC+10] Nagy Mostafa, Chandra Krintz, Calin Cascaval, David Edelsohn, Priya Nagpurkar, and Peng Wu. Understanding the Potential of Interpreter-based Optimizations for Python. Technical Report 2010-14, UCSB, 2010.

[WWS+12] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.

ViCE-UPSLA: A Visual High Level Language for Accurate Simulation of Interlocked Pipelined Processors

Dennis Klassen

University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany
dennis.klassen@uni-paderborn.de

Abstract: Simulation of processors is needed in early stages of development to reduce cost and increase quality of processor designs. Suitable simulators can be generated automatically from high-level specifications of the processor architecture. For this purpose, we have developed the domain specific visual language ViCE-UPSLA. It allows to describe pipeline based register-register, register-memory processor architectures and generates efficient simulators for such processors. In this way a variety of processors can be quickly prototyped for validation and evaluation. We have successfully used ViCE-UPSLA to model and simulate a processor with an ARM [ARM00] like architecture.

1 Introduction

In several domains, domain specific applications are developed with focus primarily on throughput. In addition to software and compiler optimization, it is necessary to optimize the processors. These optimizations can have different aims such as speed, power consumption, parallelism, real-time suitability, cost reduction, etc. However, efficiency enhancement through the development of application-specific processors is mostly in our focus.

Effects of processor optimizations must be determined with the help of simulators during the early stages of development. Developers need to estimate optimization trends from the simulation results and add promising improvements manually to the hardware design. In order to shorten development cycles, the implementation effort of simulator development has to be reduced. A development environment on the abstraction level of the processor-reference-documentation [GK83] is desirable. Then, developers would be able to immediately locate targets for optimization and test and manipulate the simulators as they need. Besides the efficient simulation of processors, the systematic validation of processors, starting from a specification on a high abstraction level, is the next step in development.

In this paper we present ViCE-UPSLA, a visual language for processor design. This visual language is inspired by the textual language UPSLA (Unified Processor Specification

LAnguage) [KLST04]. In addition to the description of an instruction set simulator, like UPSLA, ViCE-UPSLA adds the description of microarchitecture components on a high abstraction level. The visual language is characterized by the usage of typical and established terms and symbols from the processor design and architecture domain. From the processor specification, the toolchain of ViCE-UPSLA generates simulators for interlocked or non-interlocked microarchitectures automatically.

In this paper we present the concepts of the visual language. Section 3 focuses on the early stages of processor design, where an instruction set simulator is sufficient. Later, a more detailed microarchitecture simulator can be obtained, as described in Section 4. Section 5 presents the principles of our approach to generate the simulators. By looking at related works, we classify our visual language in Section 6. Some existing applications of the language are outlined next. A prospect on the language's advancement concludes the paper.

2 Concepts of the visual language

In the early stages of processor development, only abstract documents such as reference documentation or technical data sheets are available. The different background of the processor and software designers, who cooperate in the development, demands a visual language with domain specific description elements, at a comparable abstraction level to the available documents. Our visual domain specific language ViCE-UPSLA allows the specification of a processor at the level of processor-reference-documentation [arm87, pow94], with a focus on simulation and validation.

In this section, we describe the fundamental concepts of the visual language for the specification of instruction set and microarchitecture of the target processor. The language is modelled after typical processor documents with visual description components. Hence, the developer creates and speaks about a data flow graph to express a microarchitecture or to describe instruction behavior etc. From the specification, suitable simulators are automatically generated. These simulators advance the development for example via design space exploration. Another goal is the automatic static and dynamic validation of the developed processor design against its specification. The language allows to check the static consistency of the specification. Also the specification is detailed enough to generate test cases for dynamic validation.

We describe the concepts of the language using an example of the development approach. The first scenario is the development of an instruction set simulator and the second scenario is a microarchitecture simulator. After the overview of the language, we explain the parts of the specification needed to describe the different kinds of simulators.

The first scenario is typical for the beginning of the development process. Here, only an instruction simulator is needed to evaluate the requirements of the applications on the processor. The flexibility of ViCE-UPSLA allows to create and evaluate variants of a processor in a short time. Since only an instruction set simulator is needed for this, we need to describe in ViCE-UPSLA just the instructions set components. This amounts to

descriptions of the structural and behavioral properties of the instructions.

The second scenario, in later steps of the development, requires more precise processor simulators, to evaluate the correctness of the instructions in the envisioned processor architecture. Therefore we describe in ViCE-UPSLA, besides the instruction set, the microarchitecture of the processor. To describe and simulate an existing processor with a defined microarchitecture, as base line for design space exploration (DSE), the specification in ViCE-UPSLA can be adapted to capture relevant aspects of the target microarchitecture precisely. From the specification, interlocked or non-interlocked cycle accurate simulators with respect to the microarchitecture can be generated.

ViCE-UPSLA structures its processor specifications similar to the logical partitioning in the processors documents, like a processor-reference-document. This approach of the language allows the designer to transfer the contents of the reference-documentation into a ViCE-UPSLA specification nearly directly. The overall structure of the language includes four views of the fundamental components of the processor (Figure 1):

- Instruction set
- Instruction format and addressing modes
- Register set
- Pipeline architecture and Data path

The register set definition allows to describe various configurations of register banks for different architectures, including the distinction between physical and architectural registers [DHTK07]. The specifications of the register banks and single registers are referenced in the descriptions of the addressing modes and instructions. The addressing modes and the instruction formats describe the recurring structures of the instruction specifications. The instruction set consists of the entirety of all instructions and encompasses the structural and behavioral descriptions. The pipeline specifies the data path, resources and forwarding paths, respective to the instruction execution step sequence.

The components of the processor are drawn in different views and can be specified separately. In a completed processor specification, the components of the processor are linked to each other. This independent specification of linked components allows subsequent replacement or editing of processor constructs in a completed specification. For design space exploration, editing or expanding the already specified components is essential. For example, to expand the number of accessible registers for a group of instructions, with ViCE-UPSLA the developer has only to adapt the register bank size and the range of its addressing mode in one place. The specification of the microarchitecture or instruction set is not affected by this change. To ensure consistency of the specification and to support structured development, the specification components are related to each other. ViCE-UPSLA provides a number of static and dynamic validation methods to ensure the consistency of the design.

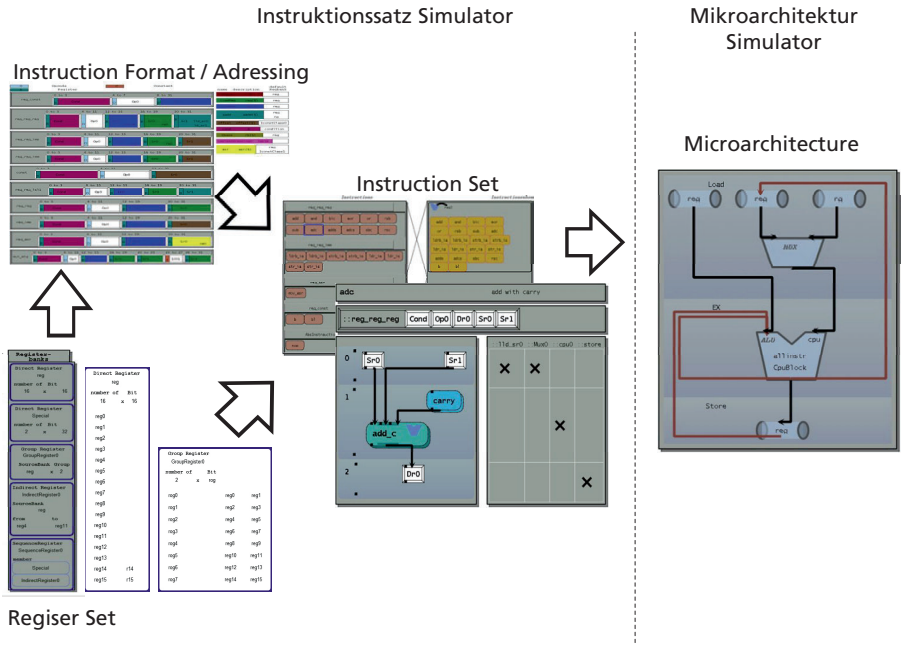


Figure 1: Schematic diagram of ViCE-UPSLA

3 Specification of an instruction set simulator

Now we present the specification steps necessary to describe an instruction set simulator with ViCE-UPSLA. To generate a simulator of an instruction set architecture, the instruction set, memory and register access specifications are needed. For this, each instruction needs a structural and behavioral description. The structural specification is needed to describe the decoding and the usage of the instruction. The execution details are specified by the behavioral description.

Instruction set For the specification of the instruction set, we have introduced the concept of abstract instructions. It supports a structured specification and classifies the instructions for validation. Besides, this approach reduces the specification effort. With the abstract instructions, we describe coherent groups of instructions and build equivalence classes. According to the pursued aspects by the developer of the processor, the equivalence classes can be determined, for example by separation of the instructions via properties like arithmetical or logical instructions or the execution duration of the instructions. The specification of an abstract instruction contains a generic structural and behavioral specifications. Each concrete instruction is derived from an abstract instruction and inherits all the specified properties of this instruction.

Instructions structure properties Most of the needed structure properties for the specification of an instruction are specified by the chosen instruction format. An instruction format combines the coding of the order and the length of the operands with the operation code. The processor specification in ViCE-UPSLA can contain instruction words with different lengths. This concept allows to describe CISC [SG79, NMEH81, HP06] instruction sets or other architectures with different instruction lengths.

The signature of the assembler notation and the machine code of the instruction is defined by the instruction format. Figure 2 illustrates an example, of the visual expression in ViCE-UPSLA, with the specified properties for the specification of the instruction format and the generated code for the visual expression. The specification of the instruction formats [HP06] in ViCE-UPSLA uses established visualizations from the domain of processor design, as shown in Figure 2. To describe the instructions of an existing processor, the individual operation codes can be defined separately for every instruction. Through the lengths of the bit fields of the instruction format the length of an instruction word of the processor is expressed exactly.

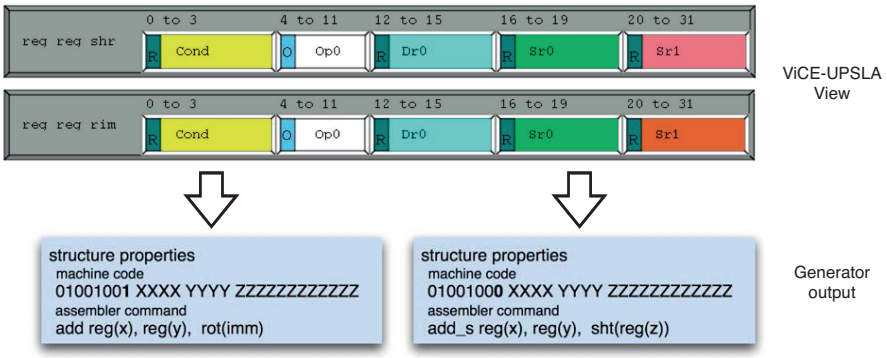


Figure 2: Two instruction formats with different addressing modes and generated code.

The additional bits to control the instruction execution or the selection of the addressing modes are interpreted in ViCE-UPSLA as operation code extensions. To specify those constructs in ViCE-UPSLA, the developer creates separate instruction formats for every configuration of the control bits. Figure 3 shows an ARM processor instruction format with two control bits I and S. Therefore with ViCE-UPSLA, the developer use four instruction formats. This allows to distinguish the instructions more clearly from each other, which is desirable for simulation and validation. For example, an existing ARM processor instruction `ADD` with and without carry is expressed in ViCE-UPSLA as two different instructions `ADD` and `ADD_C`.

Instruction Behavior The behavior of the instruction is expressed on a high abstraction level with the visualization of a data flow graph, as shown in Figure 4. The specification view combines the cycle accurate behavior and the required resources of the instruction in one diagram. The data flow graph describes the value transfer between the elements.

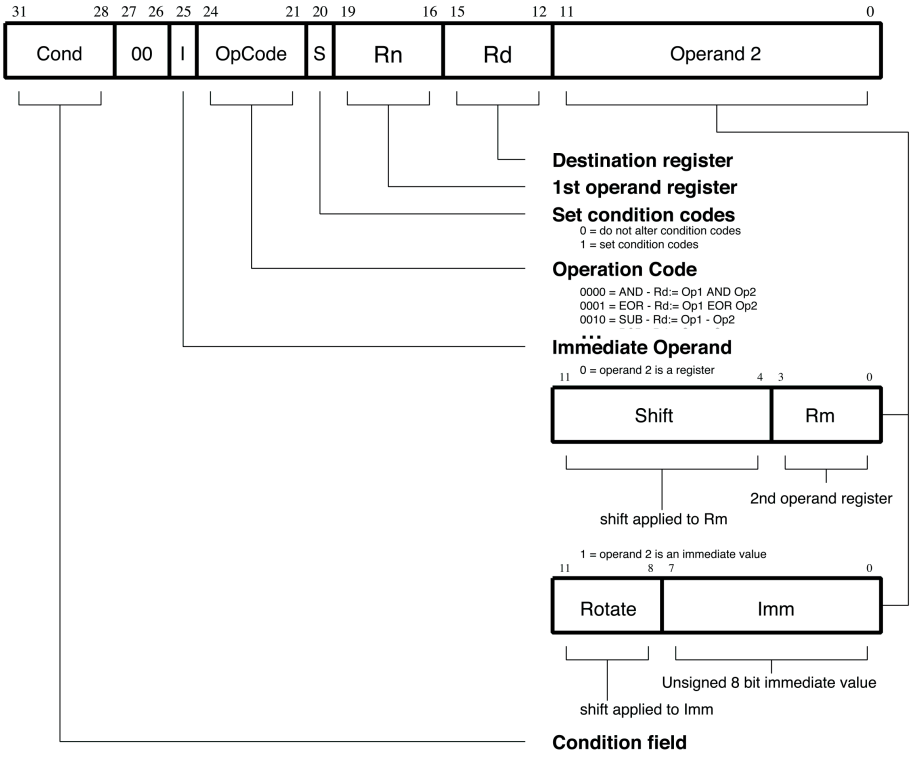


Figure 3: Excerpt of an open access ARM Data Sheet [Cop95, arm87]

During the specification process, the developer can use the operators and the interpreted operands as graph nodes, which are connected with directed edges. The operands are given by the instruction format. In addition to the interpreted operands, the implicit operands, like a `carry bit`, can be also added into the data flow graph. The data flow graph describes also the structural properties of the instruction, for example the operands' read/write direction or execution cycles of the instruction etc.

The operational description of an instruction can be specified through the cascading of operators, as shown in Figure 4. The operators specify the actions, which are applied to the input values. Determined by the developer, the complexity of a data flow graph is dependent on the granularity of the operators used. ViCE-UPSLA provides a manageable number of predefined operators. To increase the design quality, a custom set of operators can be added. For design space exploration, as well, new operators can be defined by a processor developer.

It is also possible to describe parallel execution of combined operations on different ALU's with ViCE-UPSLA. Therefore, ViCE-UPSLA supports multiple branches in the behav-

ioral description of an instruction. This allows to specify instruction sets for MIMD and SIMD architectures.

Figure 4 shows two implementations of the addition with carry (`ADD_C`) instruction, which calculates the result for `DR0` from the input operands `SR0`, `SR1` and the implicit operand `carry`.

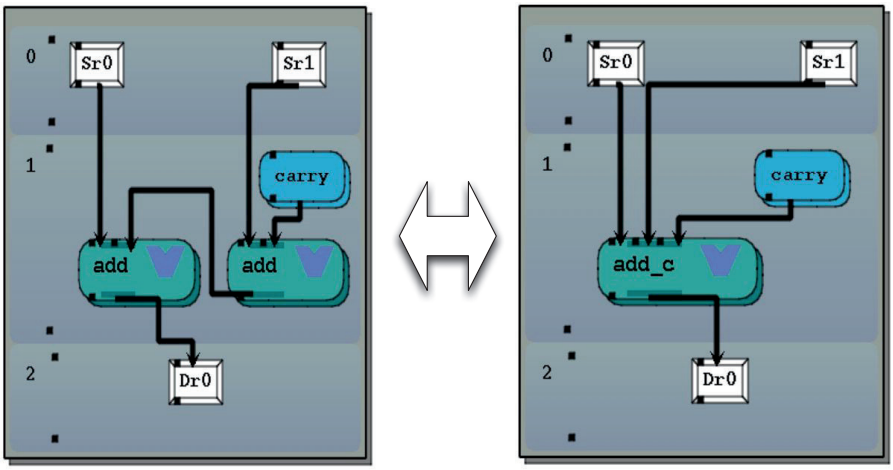


Figure 4: Instruction dataflow graph and concationation of operators

Addressing mode In ViCE-UPSLA, the specification of the instruction set is completed with the declaration of the instructions. To generate a simulator, the description of the access to the memory modules is needed, such as main memory or registers. In various architectures, like Motorola 68k [SG79, Mot92], the addressing modes describe non trivial rules for memory or registers access. The concept of the addressing modes [HP06] represents in ViCE-UPSLA the rules for the access to registers as well as memories. Figure 5 shows the concept for registers access. In the behavioral descriptions of the instructions, the interpreted operands are used. With the computation of an addressing mode, the value for the interpreted operand is provided from the register set, for the read direction, analogous for the write direction. The addressing modes are parametrised through the instruction operands from the instruction word.

An example from the ARM data sheet shows the use of interpreted and instruction operands in Figure 3. The operands `Rn`, `Rd` and `Operand 2` represent the interpreted operands. Also, Figure 3 shows two complex addressing formats `shift applied to imm` and `shift applied to Rm`. The instruction operands for the addressing modes of interpreted operand `Operand 2` are `Shift+Rm` or `Rotate+Imm`. The visualization of these specification in ViCE-UPSLA are shown in Figure 2. The addressing modes' specification contains the assignment of the bit fields and the interpreted operands. In ViCE-UPSLA, the register addresses or immediate operands can be used as instruction operands

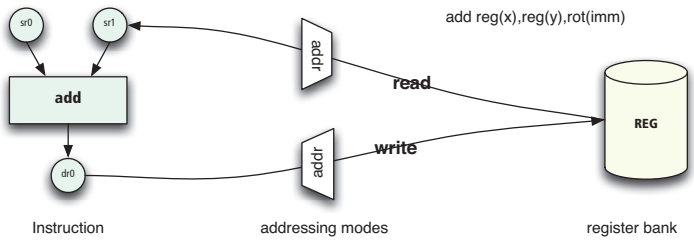


Figure 5: Register access

in the addressing modes. The register addresses refer to a register bank and the immediate operands select a value from a defined value range. With this approach, the processor developer is able to describe the access to the register set and to specify complex addressing modes with optional pre and post calculations such a Motorola 68k [Mot92] architecture addressing modes.

Register Set For simulation and validation of the processor, the register set defines the system state. During the simulation, we are able to trace the system state. To design only a instruction set simulator, a simple register specification is sufficient. ViCE-UPSLA allows to describe register set configurations for complex structures with several usage modes of the registers.

In ViCE-UPSLA, the register set of a processor includes all programmer accessible registers of the architecture [pow94, HP06]. We distinguish between two kinds of registers, the physical registers and the architectural registers. Architectural registers describe the aliasing of the physical registers. With it, different views on physical registers can be defined.

For example, the configuration of a UserMode and a SystemMode of physical registers can be specified, as shown in Figure 6. If the physical registers are `reg0-23` of register bank `reg`, registers of the `UserMode` are `usr0-15` as a slice of the registers `reg0-15`. The `SystemMode` needs two slices `reg0-7` as registers shared with the `SystemMode` and `reg16-23` as special registers. The concatenation of these define a `SystemMode`.

In ViCE-UPSLA, the architectural registers have distinct names and are characterized by the underlying sequence of physical registers. There are different constructors, to create views of physical registers to model common configurations of architectural registers. The concatenation of register banks (called SequenceRegister) or a slice of a register bank (IndirectRegister) have been used for the SystemMode as shown in Figure 6.

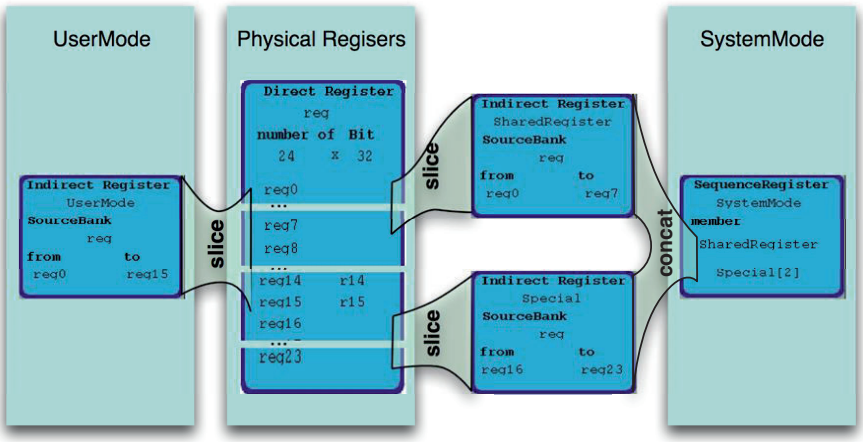


Figure 6: User Mode & System Mode

4 Specification of an interlocked microarchitecture simulator

In the previous subsection we have described the components to create an instruction set simulator. In ViCE-UPSLA, we can expand the specification from a simple instruction set simulator to a cycle and resource accurate simulator with interlocks. This approach allows to specify, simulate and validate a processors with a given microarchitecture. Like the other constructs, the specification of the microarchitecture is on a high abstraction level. The interaction between the different constructs in ViCE-UPSLA is shown in Figure 7. The microarchitecture specifies the restrictions and the coordination for the execution steps of the instructions.

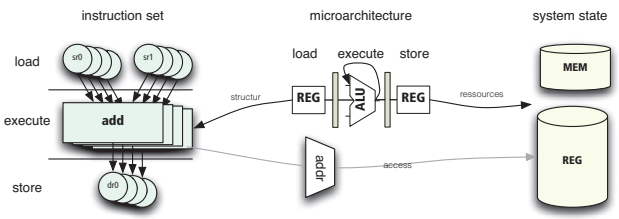


Figure 7: Construct interaction in ViCE-UPSLA

The description in ViCE-UPSLA uses the visualization of the components of the processor design domain. The basic elements of the microarchitecture are pipeline stages, which describe the sequential execution steps of the processor. Each pipeline stage groups the

function blocks, which can be utilized in the same cycle. ViCE-UPSLA supports any number of pipeline stages to describe a variety of microarchitectures.

The connected function blocks in the data flow graph of the microarchitecture express the data path of the instructions through the pipeline stages. Placed in a pipeline stage, a function block represents also a resource of this stage.

Respective to the resources in the pipeline, the behavioral instruction descriptions contain operands and operations which are linked to the pipeline resources. These are used to generate an interlocked pipelined processor simulator from specification.

To express microarchitectures like VLIW or MIMD with ViCE-UPSLA, we need to specify the corresponding behavioral description of the instructions and the microarchitecture.

The function blocks for the specification of the data flow graph in ViCE-UPSLA are:

- arithmetical logical unit (ALU)
- read- and write-ports (R/W-Ports)
- multiplexer (MUX)

The ALU describes a fixed set of instruction groups which it is able to execute. The multiplexers describe building blocks to combine data from multiple sources into a new value by calculation rules. Through multiplexers, the execution of complex addressing modes in the pipeline can be expressed. The read and write ports specify the access ports to the registers of the processor. Other dependencies between the resources are considered from the pipeline's paths.

In ViCE-UPSLA, the specification of the microarchitecture may also contain bypasses or forwardings [HP06]. To specify a bypass, the developer has only to place a bypass link between two elements of the data path. Like data paths, the bypasses can be used as directed edges in the data flow graph between any function blocks of the data flow graph. With a specified microarchitecture and instruction set, automated validation between these components is possible as well.

5 Simulator generator

With ViCE-UPSLA we generate simulators in the language C for a given processor architecture and a specific program, as shown in Figure 8. The CSim simulator generator emits the behavior of the processor while executing this specific program.

From a processor specification in ViCE-UPSLA, a code generator generates a C-framework for the constructs of the specified processor. The framework includes all steps of the instructions of the processor, split into cycle accurate actions, pipeline behavior, resource mappings etc. Also, the framework includes the computation functions for addressing modes, to prepare the values as a part of the execution of the instructions. As well as the data structure for the register set to define the system state.

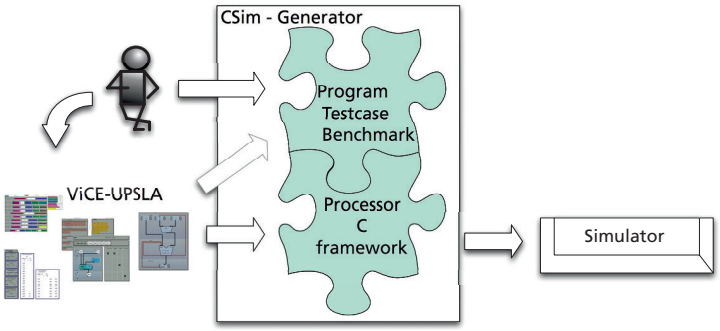


Figure 8: Simulator generator

For the generation of the simulator, the generated C-framework and the program are used. During the generation step, the assembler commands of the program are replaced by calls of C-functions from the library, which represent the processor's instructions. For simulating of the pipelined execution, the CSim generator interleaves the split cycle actions of the instruction computations according to the specified microarchitecture.

The interlocked or non-interlocked pipeline execution is part of the global specification options for a processor. With this option the developer can set, which configuration of the CSim generator is used to create the simulator. To create an interlocked processor simulator, a resource map for the instruction cycles is used. The resource map is generated from the ViCE-UPSLA specification of a processor. This map manages the instructions states for each cycle, for example currently used resources, required resources for the next cycle, released resources after the current cycle etc. The CSim generator uses the resource map to plan the execution of the next cycle for each instruction.

6 Related work

The existing languages and tools have different basic approaches to describe a processor's architecture. The basic approaches can be classified by development goals, abstraction level, expression and description possibilities. In this section we will look at different languages like the nML formalism, ViDL or Lisa to classify the ViCE-UPSLA approach.

We use the classification of Prabhat and Dutt [PD08]. Figure 9 shows the relation between structural, mixed and behavioral ADLs. In this Figure, the extremely specialized languages are shown, i.e. MIMOLA [Zim97], as a strictly structural ADL and suitable only for synthesis or validation.

On the other extreme is the language ISDL [HHD97] as a behavioral ADL suitable for simulation and compilation. However, this taxonomy is an abstract view to classify the languages. More detailed is the Y-diagram [GK83] based on the abstraction level of the

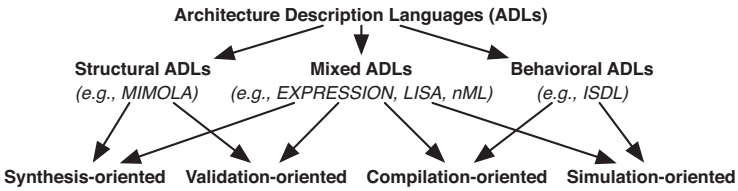


Figure 9: Taxonomy of ADLs [PD08].

description. Dieter Wecker [Wec08] describes the processors design process and explains the development steps in the Y-diagram, as shown in Figure 10. His exploration is based on the Gajski and Kuhn approach, which considers that the specification of the processor passes different abstraction levels during development and uses axes for the design of functional, structural and geometrical properties. At present, most processor or architecture description languages are mixed languages with two goals: specific construction and representation.

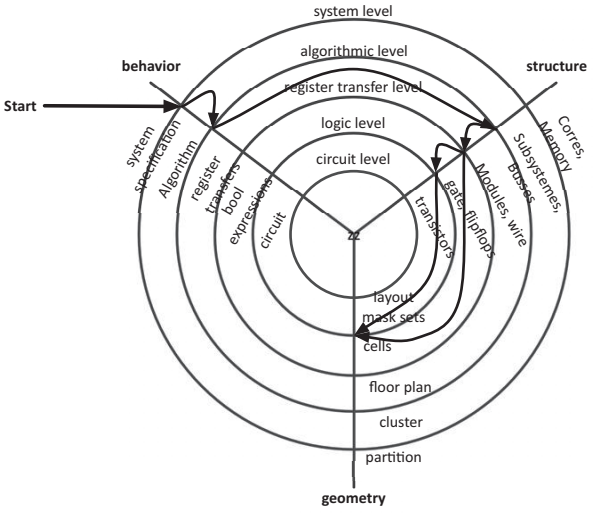


Figure 10: Y-digram [GK83, Wec08].

ViCE-UPSLA is inspired by UPSLA, introduced in Section 4 of Kastens, Le, Slowik and Thies [KLST04]. UPSLA is a textual instruction set description language, suitable for compiler backend development and simulation of the processor architectures. A ViCE-UPSLA specification describes the processor specification part of UPSLA in visual form

and expands this with the description of the processor's microarchitecture. This allows to generate a simulator as well as significant parts of the UPSLA specification and enables the static and dynamic validation of the processor specification. Thus, ViCE-UPSLA is a mixed ADL in terms of Figure 9.

The mapping from the type of the language to the development goals, shown in Figure 9, is not stringent. For example the language ViDL [Dre12] is a behavioral description language. The primary goal of the development is to generate the VHDL code, the secondary goal is simulation. Starting with an instruction set specification and the desired target clock frequency, the ViDL generator creates pipelined processor specifications in VHDL. The developer has no direct control about the number of pipeline stages or other structural processor constructs of the microarchitecture. The description in ViDL is made in textual form. In contrast to ViDL, ViCE-UPSLA combines the specification for the instruction set with the structural description of the microarchitecture. This allows to validate the specification of the instruction set against the microarchitecture.

The architecture description language xADL [BPK12] contains components, similar to ViCE-UPSLA, for the processor's description, consisting of register file, storage elements and functional units. The specification with the $adlgen$ tool supports visual specification of the data path with forwardings or pipelined execution, which can be used to generate cycle accurate simulators. Distinct from ViCE-UPSLA, the instruction set of the processor is extracted from the specification of the data path in the microarchitecture. This language follows an approach in the opposite direction of ViDL, which uses the instruction set description to generate a suitable microarchitecture. Thus, xADL is a structural ADL in terms of Figure 9.

An example for a mixed ADL is the nML [FPF95, PD08] formalism with its behavioral and structural specification, which includes execution behavior of the instructions and their encoding. The specification abstracts from a detailed description of the data path and other micro sequencing logic. This approach demonstrates very well the possibilities to implement an instruction set simulator purely with concepts of functional programming languages. However, the cognitive distance between domain specific terms of processor development and nML is rather high. The processor specification is useful to generate instruction simulators as software developer tools for programmers. Because of the missing constructs from the processor development in the description, the specification is not suitable for validation of the architecture.

Tensilica's domain-specific language TIE [Inc06] aims for comfortable design space exploration by means of processor instruction set extensions. TIE is a behavioral language with a solid structural kernel. To achieve high acceptance and reduce specification effort, Tensilica uses a fully implemented processor as a base processor architecture in its development framework. On this basis, the developer has only to concentrate on the needed extensions. This scenario is also supported by ViCE-UPSLA, with the option to let the processor developer select the desired base processor. For its visualization, Tensilica uses graphical components comparable to ViCE-UPSLA, like data flow graphs to show compiler-suggested compositions for new instructions. An additional visualization is the display of instruction formats, similar to a processor's data sheet.

The Synopsys documentation gives an overview about the language Lisa [CoW08] and its supporting toolchain. Lisa is a very flexible language, which allows to describe a diverse spectrum of processors. The visualization of the language constructs is mostly table or text based. The hierarchically structured specification requires exact knowledge about the target processor's microarchitecture right from the early stages of development. Similar to ViCE-UPSLA, behavioral specification of the instructions uses processor components expressed in the C programming language [CoW08, HL10, PD08]. In ViCE-UPSLA, visual components are used to describe the behavior, only operational descriptions given as C functions. In Lisa, the properties of the pipeline structure and the instruction's resources have to be specified separately for every instruction. This raises the development effort and makes fundamental changes to the microarchitecture more costly. In ViCE-UPSLA, we introduce different views and abstract instructions to avoid these problems.

7 Applications of ViCE-UPSLA

With the visual language ViCE-UPSLA, we have designed a processor specification language on a high abstraction level. The language is based on domain specific terms and visualizations, which are intuitive for experts in the domain of processor design. The automatically generated instruction or microarchitecture simulators can be used for instruction set exploration or validation of processors.

ViCE-UPSLA has been successfully used as the graphical user interface to replace textual UPSLA processor specifications. The language has been employed to design software defined radio (SDR) and cryptography hardware extensions for a resource efficient VLIW processor [JPD+10]. The use of ViCE-UPSLA has helped to extend the scope of design space exploration [Jun11, JSGR10, JDP+10].

With enhancements implemented in recent versions of ViCE-UPSLA, we can now generate processor simulators automatically. To evaluate the generator, we have implemented a representative subset of a user mode version of an ARM processor [arm87, ARM00]. The resulting specification utilizes all language constructs of ViCE-UPSLA. Like the original ARM architecture, our subset is a load/store architecture with 16 32-bit registers. Forwarding in the pipeline is realized through the specified bypass structure. The processor's instruction set includes arithmetical, logical, load/store and branch instructions. The execution of 32-Bit instructions with up to 4 operands is accurately done in the pipeline structure with one ALU. We use two load and one write port for the register file.

With this exemplary processor, we have demonstrated the expressiveness of our approach which allows to implement realistic processors with a multi-stage interlocked pipeline.

Future work includes more advanced case studies with different processors and matching applications programs to evaluate the simulator performance and further enhance the ViCE-UPSLA implementation.

Our next major goal is the integration of systematic methods for static and dynamic validation of the processor specification.

8 Acknowledgement

We would like to thank the project partners for the great collaboration and the German Federal Ministry of Education and Research (BMBF) for the funding of the national research project EASY-C (Enablers of Ambient Services and Systems Part C - Wide Area Coverage).

References

[arm87] *ARM Datasheet*. Acorn Computers Limited, 1987.

[ARM00] ARM Limited. *ARM Architecture Reference Manual*, arm ddi 0100e edition, 2000.

[BPK12] Florian Brandner, Viktor Pavlu, and Andreas Krall. Automatic generation of compiler backends. *Software: Practice and Experience*, 2012.

[Cop95] Copyright Advanced RISC Machines Ltd (ARM) 1995. *ARM 7TDMI Data Sheet, Open Access*, arm ddi 0029e edition, August 1995.

[CoW08] CoWare Inc. *LISA Language Reference Manual*, 2008.

[DHTK07] Ralf Dreesen, Michael Hußmann, Michael Thies, and Uwe Kastens. Register Allocation for Processors with Dynamically Reconfigurable Register Banks. *In Proceedings of the 5rd Workshop on Optimizations for DSP and Embedded Systems (ODES) held in conjunction with the 5rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2007)*, 2007.

[Dre12] Ralf Dreesen. ViDL: A Versatile ISA Description Language. *In 19th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS-19)*, 2012.

[FPF95] A. Furth, J. Van Preat, and M. Freerick. Describing Instruction Set Processors Using nML. *IEEE. The European Design and Test Conference(EDTC'95)*, pages 503–507, 1995.

[GK83] Daniel D. Gajski and Robert H. Kuhn. New VLSI Tools. *IEEE*, December 1983.

[HHD97] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An Instruction Set Description Language for Retargetability. *In Proceedings of Design Automation Conference (DAC)*, pages 299–302, 1997.

[HL10] Manuel Hohenauer and Rainer Leupers. *C Compilers for ASIPs: Automatic Compiler Generation with LISA*. Springer, 2010.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Academic Press, iv edition, September 2006.

[Inc06] Inc Tensilica. *Tensilica Instruction Extension (TIE) Language*, 2006.

[JDP+10] Thorsten Jungeblut, Ralf Dreesen, Mario Porrmann, Michael Thies, Ulrich Ruckert, and Uwe Kastens. A Framework for the Design Space Exploration of Software-Dened Radio Applications. *2nd International ICST Conference on Mobile Lightweight Wireless Systems*, 2010.

[JPD+10] T. Jungeblut, C. Puttmann, R. Dreesen, M. Porrmann, M. Thies, U. Rueckert, and U. Kastens. Resource efficiency of hardware extensions of a 4-issue VLIW processor for elliptic curve cryptography. *Advances in Radio Science.*, 8:295, – 305, 2010.

[JSGR10] Thorsten Jungeblut, Gregor Sievers, Mario Gregor, and Ulrich Rückert. Design Space Exploration for Memory Subsystems of VLIW Architectures. *5th IEEE International Conference on Networking, Architecture, and Storage, pp.*, page 377, 2010.

[Jun11] Thorsten Jungeblut. *Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren*. PhD thesis, Universität Bielefeld, 2011.

[KLST04] Uwe Kastens, Dinh Khoi Le, Adrian Slowik, and Michael Thies. Feedback Driven Instruction-Set Extension. In *Proceedings of ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, Washington, D.C., USA, June 2004.

[Mot92] Motorola Inc. *MOTOROLA M68000 FAMILY Programmer's Reference Manual*, 1992.

[NMEH81] Robert N. Noyce and Jr Marcian E. Hoff. A History of Microprocessor Development at Intel. *IEEE MICRO*, February 1981.

[PD08] Mishra Prabhat and Nikil Dutt, editors. *Processor Description Languages (Morgan Kaufmann Series in Systems on Silicon)*, volume 1. Morgan Kaufmann Publishers/Elsevier, 2008.

[pow94] *PowerPC 604: RISC Microprozessor User's Manaul*. IBM Microelectronics, 1994.

[SG79] Edward Stritter and Tom Gunter. A Microprocessor Architecture for a Changing World: The Motorola 68000. *IEEE*, February 1979.

[Wec08] Dieter Wecker. *Einführung in den Prozessorentwurf: Von der Planung bis zum Prototyp*. Expert-Verlag, August 2008.

[Zim97] G. Zimmermann. The Mimola design system. A computer aided digital processor design method. *16th Design Automation Conference*, pages 53–58, 1997.

CASM: Implementing an Abstract State Machine based Programming Language *

Roland Lezuo, Gergö Barany, Andreas Krall

Institute of Computer Languages (E185)
Vienna University of Technology
Argentinierstraße 8
1040 Vienna, Austria
{rlezuo,gergo,andi}@complang.tuwien.ac.at

Abstract: In this paper we present CASM, a general purpose programming language based on abstract state machines (ASMs). We describe the implementation of an interpreter and a compiler for the language. The demand for efficient execution forced us to modify the definition of ASM and we discuss the impact of those changes. A novel feature for ASM based languages is symbolic execution, which we briefly describe. CASM is used for instruction set simulator generation and for semantic description in a compiler verification project. We report on the experience of using the language in those two projects. Finally we position ASM based programming languages as an elegant combination of imperative and functional programming paradigms which may liberate us from the von Neumann style as demanded by John Backus.

1 Introduction

Most of the well known programming languages have a sequential execution semantics. Statements are executed in the order they appear in the source code of the program (imperative programming). The effects of a statement – changes to the program's state – are evaluated before the next statement is considered. Examples of such programming languages are assembly languages, C, Java, Python and many more. Actually there are so many languages based on the imperative paradigm that it may even feel "natural".

However, there seems to be a demand for alternative programming systems. John Backus even asked whether programming can be liberated from the von Neumann style [Bac78]. John Hughes tries to convince the world that functional programming matters [Hug89]. Hudak et al. conclude that, although not used by the masses, aspects of functional programming are being incorporated into main stream imperative languages [HHPJW07].

Functional languages describe side-effect free application of functions to their arguments which makes them presumably easier to understand. They struggle to gain real-world

*This work is partially supported by the Austrian Research Promotion Agency (FFG) under contract 827485, *Correct Compilers for Correct Application Specific Processors* and Catena DSP GmbH.

acceptance however. While not claiming to be a representative source, but merely an indicator, the popular source code repository hoster github[1] (more than 4 million repositories) lists no functional language amongst its 10 most popular ones. Actually, even assembly languages (17th most popular) are more popular than Haskell (18th most popular with approx. 16000 projects using it) according to github's language statistics.

In this paper we report on CASM a general purpose programming language based on Abstract State Machines (ASMs). ASMs were introduced by Gurevich (originally named evolving algebras) in the Lipari Guide [Gur95]. One of the core concepts of ASM (as the name indicates) is the state. Another core concept of ASM is the concept of a rule, which describes exactly how the state is changed by means of *updates* applied to the state. Application of a rule itself is side-effect free, one of the core concepts of functional programming.

The CASM language was originally designed to describe programming language semantics, a purpose for which ASMs are known to be well suited [SSB01, ZG97, Gau95, KP97, Gle04, HS00, GH93]. We perform correctness proofs in a compiler verification project using symbolic execution. Having the ASM models of a machine language at hands it suggests itself to reuse them for implementing an instruction set simulator [BHK10]. To create a fast simulator we added a type system and developed a compiler for the CASM language.

The remainder of this paper is structured as follows. Section 2 discusses previous work on ASM based programming languages, in section 3 we describe the CASM language, section 4 briefly introduces the implemented type system, section 5 explains implementation details of the interpreter and compiler, in section 6 we report on our experience using CASM in various projects and section 7 finally concludes the paper and gives directions for further work.

2 Related Work

The ASM method is well known and there have been quite a few efforts to create a widely accepted framework for ASM tools. However, most of the projects ceased to exist. One reason it seems to be so difficult to provide a generally accepted framework and language for all ASM users may be the fact that ASM's applications are so broad. On the one hand it is used to create very high level models [Bö03], and on the other it is used to model very low level aspects of hardware (like we do). The demands of the users vary greatly and this may be the reason there is quite a number of attempts to create ASM tools. In a sense this work adds to the misery, but it is driven by the very specific needs of compiler verification (symbolic execution) and simulator synthesis (very fast performance). This section tries to distinguish the CASM language from other ASM based languages.

Schmid introduced AsmGofer in [Sch01b]. AsmGofer is an interpreter for an ASM based language. It is written in the Gofer[2] language (a subset of Haskell) and covers most of the

[1] `http://github.com`

[2] `http://web.cecs.pdx.edu/~mpj/goferarc/index.html`

features described in the Lipari guide. The author notes however that the implementation is aimed at prototype modeling and too slow for performance critical applications.

Castillo describes the ASM Workbench in [Cas01]. Similar to CASM he added a type system to his language. The ASM Workbench is implemented in ML[3] in an extensible way. Castillo describes an interpreter and a plugin for a model checker, which allows to translate certain restricted classes of abstract state machines to models for the SMV[4] model checker.

Schmid also describes compiling ASM to C++ [Sch01a]. His compiler uses the ASM Workbench language as input. He proposes a double buffering technique avoiding collection update sets to increase runtime performance. There is no report of the achieved performance. CASM uses a so called *pseudo state* (more details are in section 5.1.3) to implement *update sets* efficiently.

Anlauff introduces XASM, a component based ASM language compiled to C [Anl00]. The novel feature of XASM is the introduction of a component model, allowing implementation of reusable components. XASM supports *functions* implemented in C using an *extern* keyword. CASM does not feature modularization, but can be extended using C code as well. XASM was used as the core of the gem-mex system, a graphical language for ASMs.

Farahbod designed CoreASM, an extensible ASM execution engine [FGG05]. The CoreASM project is actively maintained and early prototypes of our compiler verification proofs even used CoreASM. Unfortunately performance of the CoreASM interpreter is very poor, which ultimately lead to the development of CASM. Execution speed was increased by a factor of 3000 [LK12]. The CASM language is heavily inspired by the CoreASM language, but over time they have diverged.

3 The CASM language

In this section we introduce the most important aspects of the CASM language. The Lipari guide [Gur95] contains a formal definition of the constructs presented in this section. Also the CoreASM handbook may be a useful reference [Far] as CASM roots in the CoreASM language. In section 3.7 the most important differences between the two languages are pointed out.

3.1 State and Execution model

The state of an ASM is a so called *static algebra* over a set of universes (types) [Gur95]. These universes form a *superuniverse* X in which at least 3 distinct elements (*true*, *false* and *undef*) are defined. The state contains *r-ary* functions (mapping X^r to X) and relations

[3] http://en.wikipedia.org/wiki/Standard_ML
[4] http://www.cs.cmu.edu/~modelcheck/smv.html

(mapping X^r to *true, false*). For all *locations* of a function, unless not explicitly defined otherwise, the value *undef* is assigned. CASM programs describe a (potentially infinite) sequence of state changes by alternating calculation and application of *update sets*.

The state of a CASM program is formed by a number of *functions* (possibly 0). An example is given in listing 1.

```
function foo : -> Boolean
function bar : Int -> Int
```

Listing 1: A state

Function *foo* is a *0-ary* function, and is very similar to a global boolean variable in common programming languages. Function *bar* is a *1-ary* function, mapping an *Int* to another *Int*. This can be interpreted as an array, although there are no bounds and the function may be only partially defined. The special value *undef* will be returned for all *locations* where *bar* is undefined. In that sense a *1-ary* CASM *function* is more like a hash-map. The arguments to a function are called a *location*.

CASM programs are formulated as a set of *rules*. There is a top-level rule which will be executed repeatedly until the program terminates itself. A rule can't change the state while being executed, which implies that all calculations are side-effect free. The rule returns the changes to the state, as a so called *update set*. An *update set* is a set of *updates*, each of which describes a *function*, a *location* and the new value. *Update sets* are applied to the state whenever the top-level rule *concludes* (returns). Listing 2 shows an example of a rule (and the state it operates on).

```
function x : -> Int
function y : -> Int
function z : -> Int

rule swap = {
  x := y
  y := x
}
```

Listing 2: Parallel swap

Assuming an initial state of $\{x = 3, y = 2, z = 1\}$ the calculated *update set* of a (single) invocation of *swap* is $\{(x, 2), (y,3)\}$. The update is then applied to the state, yielding $\{x = 2, y = 3, z = 1\}$.

When a *function* is assigned different values (for the same *location*) the update set is said to be *inconsistent*. In CASM a non-revocable runtime error is raised when an update set becomes inconsistent. Listing 3 shows an example for a rule triggering a runtime error.

```
function b : -> Boolean
rule inconsistent = {
  b := true
  b := false
}
```

Listing 3: Inconsistent update set - runtime error

3.1.1 Update Rule and Function Signatures

The previous examples informally introduced the *update* rule (*:=*) and function *signatures*.

A *function signature* defines the types of the function arguments (comma-separated between *:* and ->) and the type the function maps to (after ->).

The general form of an update rule is $f(l) := v$, where f is a *function* and l a *location*. A *location* has to match the *argument types* of the *function signature*. v is an expression of the function's type which will be evaluated and assigned to the *function* at the given *location*. The state will not be changed when the *update rule* is executed, but the effects of the *update* will be added to the *update set* of the environmental rule. Only the *update rule* can change the state (or more precisely, add *updates* to an *update set*).

3.1.2 Types, Composition and Enumerations

CASM offers 3 built-in primitive types. They are *Boolean* for boolean values (*true, false*), *Int* for integer values and *String* for character strings.

An enumeration can be defined using the *enum* keyword. Each enumeration defines a new type with the same name as the enumeration. Each member of the enumeration becomes a new globally unique identifier of the enumeration's type. An example is given in listing 4.

```
enum MyEnum = { one, two, three }
function x : -> MyEnum

rule example =
  x := three
```

Listing 4: Enumeration Type

CASM offers two kinds of type composition. A *List* is a sequence of zero or more elements of equal type. The *Tuple* is a sequence with a fixed number of elements of possibly different type. CASM disallows user-provided recursive data types in the current version. An example is given in listing 5.

```
function stack : -> List(Int)
function aMapping : -> List(Tuple(String,Int))
```

Listing 5: Composition Type

3.2 Expressions, Variables and Derived Values

For *Int* types the usual set of operators is provided via built-ins. Expressions are very similar to the C programming language (without any side-effects however).

CASM does not offer a notion of local state, so there are no local variables in the common sense. There is however the possibility to bind expressions to a local name using the *let* rule. In contrast to CoreASM *let* rules are statically scoped.

```
derived d(a : Int, b : Boolean) = (a >= 3) and b
function foo : Int -> Boolean

rule bar =
  let x = 2*3 in
    let y = 3*x in
      foo(y) := d(x, true)
```

Listing 6: Nested Let Rules

Some expressions may be used extensively in a program and in more than one rule. Such expressions can be declared globally using the *derived* keyword. A *derived* accepts typed arguments, just like *functions*, and is a single expression. Listing 6 gives an example of a *derived* and nested *let* rules (assigning *true* to *function foo* at *location 18*).

3.3 Block Rule and Control Flow

The block rule, syntactically expressed by curly brackets, combines the *update sets* of all enclosed rules into one single *update set*. Each of the enclosed rules is invoked on the same state and because all rules are side-effect free the order in which the rules are invoked does not matter. The resulting *update set* is formed by applying the *union* operator on all calculated *update sets*. When merging is performed *inconsistent updates* need to trigger a runtime error. Listing 2 already made use of the block rule.

The CASM language offers an *if-then-else* rule (an example can be seen in listing 11). In the context of an ASM based language it is an *if-then-else rule* (not a statement). Thinking of *if-then-else* as a rule also helps to comprehend the semantics. *if-then-else* produces an *update set* (like any other CASM rule does). The update set is either the update set produced by the rule in the *if-branch* or the one in the *else-branch*, depending on the boolean value of the *expression* following the *if* keyword.

There also is a *switch-case* statement, with the usual semantics and an optional *default* case label.

```
rule r1(a:Int) = skip

rule r2 =
  call r1(3)

rule r3 =
  let rr = @r1 in
    call (rr)(5)
```

Listing 7: Direct and indirect call

The *call* rule invokes another rule. There are two flavors of the call rule, a direct and an indirect one. In the direct case the rule to be invoked is known statically and directly coded in the source file, while in the indirect case the rule to be invoked is calculated by an *expression* of type *RuleRef*. A *RuleRef* can be produced by the @ operator (similar to C's & operator). Listing 7 shows a direct and an indirect call (note the additional brackets

around the expression). One can also see that rules can have typed arguments. CASM's *call* rule differs from the definition given in the Lipari guide, see section 3.7 for more details.

3.4 Sequential Block Rule

Some problems can naturally be described by imperative programming. CASM supports sequential programming by means of the *seqblock* rule. Statements enclosed by a *seqblock* are executed in exactly the specified order. Additionally the state change induced by previous rules is visible to subsequent ones. The *update set* of a *seqblock* rule is calculated by subsequently merging the *update sets* of the enclosed rules. Two updates to the same *function* and *location* from different rules do not conflict however, the later overwrites the previous one. Effectively the *update set* describes what would have happened if the rules were applied using a sequential semantics.

It is important to note that the state is not actually changed. The rules are evaluated using the state which would result from applying the previous update sets. A temporary state is created and each calculated update set is applied to it. This temporary state is discarded at the end of the *seqblock*.

Listing 8 shows an implementation of swap using a *seqblock* rule. Please note the use of the temporary variable *t*, because the update to *x* is visible to the second *update rule* (compare to listing 2). Again assuming an initial state of $\{x = 3, y = 2, z = 1\}$ the *update set* of the first *update rule* is $\{(x,2)\}$. This update set is applied to the initial state resulting in the temporary state $\{x = 2, y = 2, z = 1\}$. The second *update rule* results in the update set $\{(y,3)\}$. Applying to the temporary state the gives $\{x = 2, y = 3, z = 1\}$, this state is discarded however. Merging the update sets gives the update set of the *seqblock* rule itself, it is $\{(x,2), (y,3)\}$. This final update set will be applied to the state when the *swap* rule concludes.

```
function x : -> Int
function y : -> Int
function z : -> Int

rule swap =
  let t = x in
    seqblock
      x := y
      y := t
    endseqblock
```

Listing 8: Sequential swap

3.5 Forall and Iterate

The *forall* rule is used in combination with an iterate-able expression (e.g. a range of integers, an enumeration). For each element of the iterate-able expression the rule given in the body is invoked. The *update sets* produced by the body are understood to be executed in parallel. An example is given in listing 9 where it is used to create a list of 4 elements.

```
function x : -> Int

rule create =
  forall i in [0..3] do
    x(i) := i
```

Listing 9: Forall

The *iterate* rule on the other hand repeatedly invokes its rule body until the produced *update set* is empty. Each invocation is understood to be executed in a sequential manner, so each rule is invoked using the *temporary state* produced by the previous iteration. An example is given in listing 10 where the rule is used to atomically perform a fold operation (using addition) on a list. The update set returned by the *iterate* rule is {*(i,10),(f,45)*}.

```
function a : Int -> Int initially { 0->0, 1->1, /* skipped */ , 9->9 }
function i : -> Int initially { 0 }
function f : -> Int initially { 0 }

rule fold =
  iterate
    if i < 10 then {
      i := i + 1
      f := f + a(i)
    }
```

Listing 10: Iterate

3.6 Stacks and Lists

There are rules and built-in functions which can be used with *List* types. Rules *pop* and *push* are used to implement stacks. The built-in functions *cons*, *peek* and *tail* construct and consume lists. There also is a *nth* function to access the nth element of a list (or tuple). These functions are (parametric) polymorphic in their nature and need to be handled correctly by the type system. Listing 11 gives an example.

```
function list : -> List(Int)

rule foo =
  seqblock
    push 3 into list
    if peek(list) != 3 then assert false
    let x = nth(list, 1) in list := cons(x, list)
  endseqblock
```

Listing 11: List built-ins and rules

3.7 Differences to other ASM based languages

CASM differs in two major aspects from other ASM based programming languages (i.e. CoreASM).

The Lipari guide defines rule arguments to be passed-by-name. Passing by-name has some interesting features, but is difficult to be implemented efficiently [BG93]. In CASM the semantics of the *call* rule specify arguments to be passed by-value. Please note that pass-by-value is semantically equivalent to pass-by-name when all arguments are constants. Therefore the restricted *call* rule of CASM can be simulated by evaluating all argument expressions (e.g. using a *let* rule) before invoking an unrestricted *call*. We merely enforce this policy on the CASM programmer by performing this evaluation before invoking the called rule.

The other difference is the scope of variables introduced by the *let* rule. CoreASM installs variable names into an environment passed to rules invoked by a *call* rule. So listing 12 is valid in CoreASM, but not in CASM. Environments are not passed to rules invoked by a *call* rule in CASM. The main reason for this is that the type inference and type checking would be unable to handle this.

```
function y : -> Int

rule callee = y := x
rule caller =
  let x = 3 in
    call callee()
```

Listing 12: Only valid in CoreASM

4 CASM Type System

CASM is a static[5], strongly typed language and no implicit type conversion is performed. To reduce the often redundant notation of types the programmer is allowed to omit the type if it can be deduced automatically. CASM only demands types for the arguments of *functions*, *rules* and *derived* as well as for a *function's type*. Optionally the programmer can provide type information for the type of a *derived* and a named expression type bound via a *let* rule. This may improve readability of the source code and may be needed to guide the type inference system in some corner cases. These corner cases often result from the special value of *undef* which is compatible to every type.

```
function assign : -> List(Int)

rule foo =
  let uList : List(Int) = undef in
    let uElem = nth(1, uList) in
      assign := [ uElem ]
```

Listing 13: Type Annotation needed

[5] except indirect calls

Listing 13 shows an example the CASM type system implementation can not handle without annotation. The first *let* binds the name *uList* to the value *undef*. The second *let* binds the name *uElem* to the *nth* (1st in this case) element of a list or tuple. *nth* (a built-in function) returns *undef* if any of its arguments is *undef*. *uElem's* type could not be computed (locally) when no additional type information would be provided by the programmer. Although limitations in the implementation of the type system exist, they very rarely occur in real world programs.

To specify the argument types of *rules* and *deriveds* may be superfluous, but increases readability and documentation. But especially for *derived* it prevents parametric polymorphism, which may be a useful feature after all. We are considering to change this in a future version of CASM.

5 Interpreter and Compiler

We have developed an interpreter and a compiler for CASM. The interpreter is capable of concrete and symbolic execution of CASM programs. We have also developed a compiler generating C++ code.

The CASM interpreter is a simple abstract syntax tree (AST) interpreter. For creating the parser traditional compiler tools like lex and yacc have been used. Type inference and type checking is performed on the AST. The program is rejected if any types can't be calculated or any type mismatch is detected.

During symbolic execution some (or all) values of the state can hold symbolic instead of concrete values. Evaluation of operators then depends on whether all operands are concrete values or if there is at least one symbolic one. As long as all operands have concrete values the operator is evaluated as usual. But if there is at least one symbolic operand the operator itself returns a new symbol. This returned symbol is linked to the fact that it had been calculated by applying the operator to the specific operands. E.g. an addition of the symbol *s3* and the concrete value *23* will result in a new symbol *s4*. *s4* will be linked to the fact *s4 := s3 + 23*. Should *s4* ever be used as an operand a new symbol will be created as result, linked to the fact that *s4* was used as an operand. Any symbol appearing as the result of a program can that way be traced back to an initial symbol provided as input to the program.

Things get interesting when control flow branches on a symbolic value [Kin76]. For the sake of brevity we only consider *if-then-else* rules here. The *conditional expression* must be of *Boolean* type in CASM, so it can only have two possible values (*true, false*). When the boolean value is symbolic, both possible values need to be considered. The program *forks* assuming the *expression* to be *true* in one case and *false* in the other one. These assumptions become part of the facts known about the symbols. The sum of all facts learnt about symbols along a path of execution is called path condition. Path conditions can be used to automatically generate test cases [VPK04]. The CASM interpreter does not directly support this, but it can easily be implemented by using the generated trace files.

5.1 Compilation scheme and efficient runtime

The typed AST is also used to compile the CASM program to C++. Our current CASM compiler implementation performs only a limited set of optimizations to keep the generated C++ files small. The runtime system makes heavy use of C++ template mechanism and inlining and hence the generated machine code is therefore quite compact resulting in satisfying performance for common CASM code patterns.

5.1.1 Efficient Memory Allocation

During evaluation of a rule a number of updates and update sets have to be generated dynamically. The lifetime of these objects is limited however. All update sets and the updates they contain can safely by deleted when the top-level rule has concluded and all updates have been committed to the state. We therefore allocate these objects in a dump memory area. When the top-level rule concludes, the dump memory is reset and all objects it contained are invalid. New updates are allocated overwriting the old ones. This technique reduces overhead for dynamic memory allocation to almost zero.

5.1.2 Optimization for very small update sets

For the workloads we operate on we observed that most update sets are very small ($<= 4$ in most cases). Our current implementation for update sets therefore use a small number (4) of pre-allocated slots. Only after all pre-allocated slots of an update set are occupied a hash-map is used to store any additional updates part of the update set. A hash-map needs to be used as update sets are frequently tested for membership (to detect conflicting updates).

5.1.3 Pseudo State

Handling of the *update sets* is crucial for the performance of the compiled code. CASM uses hash-maps to implement *update sets*. The underlying assumption justifying this design decision is that *update sets* are small and the global state is large. A concept called *pseudo state* is used to realize temporary states needed to implement *sequential* execution semantics (see section 3.4). When reading a state *function* from within a *seqblock* the *update set* is queried first. If there has been an update to that *function* (and *location*) by a preceding rule, the value from the *update set* is returned. Otherwise the value is read from the global state. All rules return the update set produced according to their semantics (described in section 3). Using this mechanism the update sets only needs to be applied to the state when the top-level rule *concludes*. Merging and querying hash-maps is efficient as long as they are reasonably small.

6 Evaluation of the CASM language

6.1 Hardware modeling

We successfully used CASM in two projects to model CPU architectures and will briefly report the results here. Details can be found in the cited papers.

One project focused on fast design space exploration synthesizing cycle-accurate simulators from a CASM model of a microprocessor. The microprocessor model is proven to be coherent to the CASM specification of its instruction set architecture. We were able to model the instruction set and two variants of pipelined MIPS processors in just a few hundred lines of code. This is in size similar to a MIPS model formulated in a specialized hardware description language and demonstrates the expressiveness of the CASM language. The synthesized simulator is capable of executing benchmark programs of the SPECInt suite with a very satisfying peak performance of 1 MHz. This roughly translates into 15 million (basic) CASM rules to be executed per second. More details are reported in [LK13].

In a compiler verification project CASM is used to model a complex (non-interlocking) DSP processor. Combining parallel and sequential execution modes emerged as the crucial feature to describe cycled circuits. All hardware blocks are described in a parallel execution block, whereas their internal operations are described by a sequential execution block. We primarily use symbolic execution to perform simulation proofs in a translation validation approach. Some details are reported in [LK12].

6.2 Functional and Imperative programming style

In the introduction it was claimed that ASM based languages in a way combine imperative and functional programming styles. This property of the CASM language proved to be very useful in our experience. In this section we want to point out this property giving an illustrative example.

Börger and Bolognesi give a recursive ASM version of quicksort in [BB03]. Their implementation is very short and concise, like it is in most functional languages, but is not in-place as well. Imperative implementations swap elements directly (in-place) in the array avoiding copies of the whole input data. Functional languages need to construct a new list containing the resulting array (they can not destroy the input data) which induces $\mathcal{O}(n)$ additional space requirements. ASM based languages calculating quicksort in one computation step return an update set describing the new state of the array. This update set also uses $\mathcal{O}(n)$ additional space. We present an in-place, non-recursive version of quicksort.

```
function stack : -> List(Tuple(Int, Int)) initially { [] }
function array : Int -> Int
function p : -> Int initially { undef }
function l, r, ll, rr, pivot : -> Int
function need_pop, need_partition : -> Boolean
```

Listing 14: Quicksort (state)

```
rule partition_one_step =
seqblock
  iterate
    if array(ll) < array(pivot) then
      ll := ll + 1
  iterate
    if array(rr) >= array(pivot) and
       ll < rr then
         rr := rr - 1
  if ll < rr then {
    array(ll) := array(rr)
    array(rr) := array(ll)
  }
  else
    need_partition := false
endseqblock

rule partition =
if pivot = undef then {
  pivot := r
  rr := r - 1
  ll := l
  need_partition := true
}
else if need_partition then
  call partition_one_step
else {
  p := ll
  if pivot != ll then {
    array(pivot) := array(ll)
    array(ll) := array(pivot)
  }
}
```

Listing 15: Partition

```
rule quicksort_one_step =
  if p = undef then
    call partition
  else seqblock
    if l < p-1 then
      push [l,p-1] into stack
    if p+1 < r then
      push [p+1,r] into stack
    need_pop := true
  endseqblock

rule quicksort =
  if need_pop then {
    let top = nth(stack, 1) in
      if top != undef then {
        stack := tail(stack)
        l := nth(top, 1)
        r := nth(top, 2)
        pivot := undef
        p := undef
        need_pop := false
      }
      else
        program(self) := undef
  }
  else
    call quicksort_one_step

rule init = {
  push [0,SIZE] into stack
  need_pop := true
  program(self) := @quicksort
}
```

Listing 16: in-place, non-recursive Quicksort

Listing 14 shows the state needed to perform the algorithm. It uses a *stack* to keep track of the parts of the *array* still to be sorted. The *quicksort* rule pops a new part to be sorted of the *stack* and stores the left and right indices into *l* and *r*. Line 48 tests if there is still further work to do and if so lines 49-45 contain the necessary initializations. The CASM idiom in line 57 terminates the whole computation. As long as a part of the array still needs to be sorted the rule *quickstep_one_step* will be executed. The rule *partition* will be called until a partition has been found. If the remaining parts are not trivial small they are pushed onto the *stack* and a new part needs to be popped from the stack *need_pop*.

The *partition* rule initially determines a pivot element (the last element of the part to be

sorted here) and either swaps two elements of the array (rule *partition_one_step*) or swaps the pivot element to its final position p. Because the *partition_one_step* rule concludes after swapping two elements of the array the resulting update set is size bound and not dependent on the input data. Otherwise the update sets would need up to $\mathcal{O}(n)$ memory (i.e. on input $y_0, y_1, \ldots y_n, x_0, x_1, \ldots x_m, p \ : y_i < p \wedge x_j > p \ : 0 \leq i \leq n, 0 \leq j \leq m$).

To keep the observable computations small and the program simpler *partition_one_step* utilizes the *iterate* rule in lines 4 and 7 when searching for elements to be swapped. Otherwise it would need to keep track of the search similar to *need_partition*.

An ASM based language shares many features of a functional programming language. Large parts of the program are executed side-effect free, and while executing them the global state is read-only. One could think of the state as an explicit argument to each rule, the returned *update set* would then be the result of a functional application. The state only changes between invocations of the top-level rule, so the assumed implicit state argument changes for the next invocation. In that sense each single application of an ASM top-level rule is functional.

By combining parallel and sequential execution modes a programmer can chose the granularity of the computations steps seen by an observer of the program. This can, as demonstrated, be used to implement an in-place version of quicksort while basically programming a functional style.

The beauty of ASM based languages is the clear separation of invoking rules and applying changes to the state. They closely resemble what Backus called an applicative state transition system (AST system), on a much smaller scale though. What he calls a *formal system for functional programming* (FFP system) correspond to a rule and his *SYSTEM* state is just the state of the program.

7 Conclusion and Further Work

In this paper we presented CASM, a general purpose programming language based on the abstract state machine (ASM) method. We presented core features of the language and described how we are able to efficiently compile CASM to C++. The CASM compiler was successfully used to generate an instruction set simulator for the MIPS architecture capable of executing SPECInt benchmark with a peak performance of 1 MHz. We also developed an interpreter capable of symbolic execution which is successfully used in an ongoing compiler verification project.

The CASM language in a way combines functional and imperative programming aspects. We found this feature very useful and convenient and tried to showcase it using a small example.

While using the language common programming pattern arise which lead to new *rules* being implemented and new built-in functions being added. We are also working on a faster compilation and runtime implementation further increasing the performance of generated simulators.

References

[Anl00] Matthias Anlauff. XASM- An Extensible, Component-Based Abstract State Machines Language. In Yuri Gurevich, PhilippW. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 69–90. Springer Berlin Heidelberg, 2000.

[Bö03] Egon Börger. Abstract State Machines: A Method for High-Level System Design and Analysis, 2003.

[Bac78] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[BB03] Egon Börger and Tommaso Bolognesi. Remarks on Turbo ASMs for Functional Equations and Recursion Schemes. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines 2003*, volume 2589 of *Lecture Notes in Computer Science*, pages 218–228. Springer Berlin Heidelberg, 2003.

[BG93] John Bergin and Stuart Greenfield. Teaching parameter passing by example using thunks in C and C++. *SIGCSE Bull.*, 25(1):10–14, March 1993.

[BHK10] Florian Brandner, Nigel Horspool, and Andreas Krall. DSP instruction set simulation. In Shuvra S. Bhattacharyya, Ed Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 679–705. Springer, August 2010.

[Cas01] Giuseppe Del Castillo. The ASM Workbench - A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models Tool Demonstration. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 578–581, London, UK, UK, 2001. Springer-Verlag.

[Far] Roozbeh Farahbod. CoreASM Language User Manual.

[FGG05] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An extensible ASM execution engine. In *Proc. of the 12th International Workshop on Abstract State Machines*, pages 153–165, 2005.

[Gau95] Thilo S. Gaul. An Abstract State Machine specification of the DEC-Alpha Processor Family, 1995.

[GH93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini, and Michael M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.

[Gle04] Sabine Glesner. An ASM Semantics for SSA Intermediate Representations. In Wolf Zimmermann and Bernhard Thalheim, editors, *Abstract State Machines 2004. Advances in Theory and Practice*, volume 3052 of *Lecture Notes in Computer Science*, pages 144–160. Springer Berlin / Heidelberg, 2004.

[Gur95] Yuri Gurevich. *Evolving algebras 1993: Lipari guide*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[HS00] James K. Huggins and Wuwei Shen. The Static and Dynamic Semantics of C, 2000.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1989.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[KP97] Philipp W. Kutter and Alfonso Pierantonio. The Formal Specification of Oberon. *Springer Journal of Universal Computer Science*, 3(5):443–503, 1997.

[LK12] Roland Lezuo and Andreas Krall. A Unified Processor Model for Compiler Verification and Simulation Using ASM. In *ABZ*, pages 327–330, 2012.

[LK13] Roland Lezuo and Andreas Krall. Using the CASM Language for Simulator Synthesis and Model Verification. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, pages ?–? ACM, 2013.

[Sch01a] J. Schmid. Compiling Abstract State Machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001. `http://www.jucs.org/jucs_7_11/compiling_abstract_state_machine`.

[Sch01b] Joachim Schmid. Introduction to AsmGofeer, 2001.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. Java and the Java Virtual Machine - Definition, Verification, Validation, 2001.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[ZG97] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, 3:504–567, 1997.

MD^2-DSL — eine domänenspezifische Sprache zur Beschreibung und Generierung mobiler Anwendungen

Henning Heitkötter, Tim A. Majchrzak, Herbert Kuchen

Institut für Wirtschaftsinformatik
Universität Münster
Leonardo-Campus 3, 48149 Münster
{heitkoetter,tima,kuchen}@ercis.de

Abstract: Entwickler mobiler Anwendungen, sogenannter Apps, stehen einer heterogenen Landschaft an mobilen Plattformen gegenüber, die sich hinsichtlich ihrer Programmierung stark unterscheiden. Häufig sollen zumindest die weit verbreiteten Betriebssysteme iOS und Android unterstützt werden. Dabei sollen Apps dem nativen, plattformtypischen Aussehen und Verhalten genügen oder zumindest nachempfunden werden. Bestehende Cross-Plattform-Lösungen im mobilen Umfeld unterstützen letztere Anforderung nicht, so dass Entwickler oftmals gezwungen sind, dieselbe App parallel für mehrere Plattformen nativ zu entwickeln. Dies erhöht den Entwicklungsaufwand erheblich, zumal die Implementierung auf einem niedrigen Abstraktionsniveau erfolgt. Dieser Beitrag stellt das Framework MD^2 und die domänenspezifische Sprache MD^2-DSL vor, die es ermöglicht, Apps mit vorwiegend geschäftlichem Hintergrund auf einem gehobenen Abstraktionsniveau prägnant zu beschreiben. Der modellgetriebene Ansatz MD^2 generiert aus den textuellen MD^2-DSL-Modellen iOS- und Android-Apps. Die Sprache MD^2-DSL enthält Konstrukte zur Abbildung und Umsetzung typischer Anforderungen an Apps und stellt somit eine Alternative zur wiederholten Implementierung einer App auf verschiedenen Plattformen dar.

1 Einführung

Mit der steigenden Beliebtheit von mobilen Endgeräten wie Smartphones und Tablet-Computern rücken auch die Anwendungen für diese in den Fokus. Mobile Applikationen – kurz Apps – verleihen den Geräten Vielseitigkeit und erlauben, die Leistung der Hardware auszunutzen. Die Nachfrage nach Apps ist groß. Aus diesem Grund beschäftigen sich auch Unternehmen zunehmend mit der Entwicklung von Apps.

Problematisch ist die Fragmentierung des Marktes: mit Android, Blackberry, iOS, Symbian und Windows Phone gibt es fünf Plattformen, die signifikante Anteile haben [Gar12]. Aufgrund erheblicher Unterschiede etwa bezüglich Schnittstellen und Programmiersprachen muss für jede Plattform einzeln entwickelt werden, wenn Apps eine hohe Leistung bieten und eine native Benutzerführung aufweisen sollen. Gerade letztere ist für Unternehmen bedeutsam, wie wir durch die Arbeit mit unseren Praxispartner erfahren haben.

Modellgetriebene Softwareentwicklung (MDSD) [SV06] wird in Unternehmen mittler-

weile erfolgreich eingesetzt. Methoden und Werkzeuge werden durch Forschungstätigkeiten stetig verbessert. Die Grundidee ist dabei, ein fachliches Problem bzw. die dazu intendierte Lösung durch Software als Modell zu beschreiben. Aus diesem lässt sich Code generieren – bei geeigneter Anwendung auch für mehrere Plattformen. Wir haben daher mit MD^2 ein modellgetriebenes Cross-Plattform-Framework für die Implementierung mobiler Applikationen entwickelt. Apps werden in der eigens entwickelten domänenspezifischen Sprache MD^2-DSL spezifiziert und zu nativem Code transformiert.

Einen Überblick über den MD^2-Ansatz gibt bereits [HMK13]. Hier werden auch die verwendeten Generatoren genauer erläutert. Das vorliegende Paper fokussiert auf die zugehörige Domain-Specific Language (DSL) und stellt diese detailliert dar, während die Generatoren nur kurz angesprochen werden.

Dieser Beitrag ist wie folgt aufgebaut: Kapitel 2 gibt eine Einführung in domänenspezifische Sprachen. In Kapitel 3 beschreiben wir die Architektur unseres modellgetriebenen Ansatzes MD^2. Als Hauptteil des Beitrags stellt Kapitel 4 MD^2-DSL vor und erläutert, wie mit der Sprache Apps spezifiziert werden. Daran schließt sich in Kapitel 5 ein kurzer Einblick darein an, wie unser Ansatz aus den textuellen Modellen iOS- und Android-Apps generiert. Kapitel 6 diskutiert unseren Ansatz und insbesondere MD^2-DSL vor dem Hintergrund relevanter Literatur und vergleichbarer Ansätze im Umfeld von Programmiersprachen. Das Fazit in Kapitel 7 schließt den Beitrag ab.

2 Domänenspezifische Sprachen für die Anwendungsentwicklung

Domänenspezifische Sprachen (DSL) sind formale Sprachen, die für die Nutzung in einem bestimmten Problemkontext – der Domäne – entworfen wurden. In Anlehnung an [Fow10, Kap. 2.1] lassen sie sich grob wie folgt charakterisieren:

- DSLs werden von Menschen genutzt, um Computer für bestimmte Aufgaben zu instruieren. Viele DSLs können als Programmiersprachen mit einem sehr hohen Abstraktionsniveau angesehen werden.
- Jede DSL ist zugeschnitten auf eine bestimmte *Domäne*, die durch technische Vorgaben (z.B. Verwendung eines bestimmten Frameworks) und/oder inhaltliche Aspekte (z.B. E-Shop) bestimmt ist.
- Eine DSL ist typischerweise nicht Turing-mächtig.

DSLs lassen sich i.d.R. (auch von wenig technikaffinen Anwendern) leicht erlernen. Sie bieten eine kompakte Darstellung sowie eine hohe Ausdruckskraft in Bezug auf die Problemdomäne.

In unserem Fall liegt die Verwendung einer DSL nahe: es sollen so genannte *Business Apps* entwickelt werden, also mobile Applikationen, die einem geschäftlichen Zweck dienen. Insbesondere sind dabei datengetriebene Applikationen gemeint, die aus Formular-basierten Eingabe- und Ausgabemasken aufgebaut sind. Typischerweise implementieren

sie die *CRUD*-Funktionalität (Create, Read, Update, Delete) und greifen auf gerätespezifische Funktionen wie GPS zu. Diese Domäne ist einerseits im Umfang beschränkt, bildet aber andererseits fast alle für Unternehmen relevanten Szenarien ab.

DSLs kommen häufig bei modellgetriebener Softwareentwicklung zum Einsatz. Hierbei wird das zu erstellende Programm zunächst als textuelles oder grafisches Modell beschrieben. Anschließend folgt eine Reihe von Transformationsschritten, die schließlich zur Generierung von Quelltext führen. Je nach Ansatz und Vorgehensweise kann dieser Quelltext dann noch modifiziert werden; idealerweise stellt er aber bereits das Zielprogramm dar. DSLs bieten sich an, um kompakte aber gleichzeitig semantisch *reiche* Modelle zu erzeugen. Da mit dem Modell von der Zielplattform abstrahiert werden kann, bietet sich dieser Ansatz insbesondere für die Cross-Plattform-Entwicklung an.

Der Wunsch, Cross-Plattform-Entwicklung wesentlich zu vereinfachen, und die Eignung der Domäne Business Apps führten zu unserer Entscheidung, MD^2 als modellgetriebenes Framework mit der eigenen DSL MD^2-DSL zu implementieren. Dazu wurden zunächst mit Partnern in der Industrie typische Anforderungen an Business Apps herausgearbeitet. Anschließend folgte die Entwicklung eines Prototypen. Wir unterstützen zunächst die Code-Generierung für Tablets unter Android und iOS. Durch diese nicht-konzeptionelle Einschränkung kann ein Erfahrungsschatz aufgebaut werden, bevor wir die arbeitsintensive aber wenig Erkenntnisse liefernde Erweiterung für zusätzliche Plattformen angehen.

3 Konzepte und Architektur von MD^2

Im Folgenden wird kurz in die Grundkonzepte und Architektur von MD^2 eingeführt. Details finden sich in [HMK13]. Die Entwicklung von Apps mit MD^2 erfolgt in drei Schritten. Dabei erfordert nur der erste manuelle Arbeit durch den Entwickler. Dieser beschreibt als erstes die App als textuelles Modell. Im zweiten Schritt wird dieses Modell von einen Code-Generator verwendet, um plattformspezifischen Quelltext sowie zusätzliche Strukturelemente wie etwa XML-Konfigurationsdateien zu erzeugen. Es kommt pro unterstützter Plattform ein eigener Code-Generator zum Einsatz, da die Heterogenität der Plattformen einen plattformunabhängigen Zwischenschritt in der Transformation wenig sinnvoll macht: die zu erwartenden Synergien sind geringer als der Komplexitätsanstieg.

Als drittes muss der Quelltext zu einer App kompiliert werden. Dieser Schritt wird derzeit durch den Entwickler angestoßen, ist aber aufgrund der Nutzung der entsprechenden Entwicklungsumgebungen (Android Developer Tools bzw. Xcode für iOS) komplett automatisiert. Die erzeugten nativen Pakete können auf Endgeräte ausgebracht oder im Simulator debuggt werden. Da die dritte Phase ebenfalls komplett in den Arbeitsablauf integriert werden kann, ist die einzige herausfordernde (und zeitaufwendige) Aufgabe des Entwicklern das Schreiben von MD^2-DSL-Code. Während die ersten beiden Phasen durch MD^2 abgedeckt sind, erfolgt in der dritten Phase der Rückgriff auf die Werkzeuge der Plattformentwickler. Unser Framework stellt für alle drei Schritte Hilfsmittel zu Verfügung. Insbesondere die textuelle DSL MD^2-DSL ist ein wesentlicher Beitrag, da sie auf Business Apps zugeschnitten ist (siehe nächstes Kapitel). Im Rahmen von MD^2 wird auch eine

Entwicklungsumgebung bereitgestellt, die mit den üblichen Funktionen wie Syntaxhervorhebung, Inhaltsassistenz und Validatoren die Entwicklung maßgeblich vereinfacht.

Die Code-Generierung wird automatisch gestartet, sobald das Modell gespeichert wird. Dabei erzeugt der Parser zunächst die abstrakte Syntax des Modells und stellt diese für die weiteren Schritte der Generierung bereit. Ein Preprocessing bereitet das Modell vor. Danach wird das Modell durch die plattformspezifischen Generatoren traversiert, die sukzessive Quelltext erzeugen. Neben den Quelltexten – Java für Android und Objective-C für iOS – werden weitere Dateien generiert. Hierbei handelt es sich vor allem um XML-Dokumente, die z.B. Elemente der Benutzerschnittstelle spezifizieren. Hinzu kommen Konfigurationsdateien für die Entwicklungsumgebungen. Die generierten Apps werden mit statischen Bibliotheken gepackt, die häufig genutzt Funktionen bündeln.

Durch einen weiteren Code-Generator wird ein Server-Backend erzeugt. Es basiert auf dem Datenmodell der App und ist auf JavaEE-Applikationsservern lauffähig. Das Backend ist zwar in der bereitgestellten Form ausführbar, soll aber als Blaupause dienen, um die serverseitige Geschäftslogik zu implementieren bzw. unternehmensinterne Systeme anzubinden. Somit bleiben Apps schlank; bereits zur Verfügung stehende Dienste lassen sich nutzen. Dies entspricht auch dem Wunsch der meisten Unternehmen.

MD^2-DSL und auch die generierten Apps genügen dem Model-View-Controller (MVC) Entwurfsmuster. Folglich ist das textuelle Modell in *Model*, *View* und *Controller* aufgeteilt (vgl. Abb. 3, 5 und 7). In Apps wird der Controller durch ein Ereignis-System realisiert, das die Benutzerinteraktion abbildet sowie auf interne und externe Ereignisse reagiert.

4 MD^2-DSL

Dieses Kapitel stellt die domänenspezifische Sprache MD^2-DSL vor. Das erste Unterkapitel beschreibt, wie die Sprache auf Basis typischer Anforderungen mobiler Anwendungen entwickelt wurde. Es beschreibt ferner die Komponenten der Sprache und ihre Implementierung. Die folgenden Unterkapitel beschreiben die Elemente zur Datenmodellierung, zur Beschreibung der Benutzeroberfläche und zur Spezifikation der Kontrolllogik im Detail.

4.1 Entwurfsüberlegungen

Die Sprache MD^2-DSL wurde ausgehend von funktionalen Anforderungen aus der App-Entwicklung entworfen. Der Funktionsumfang der Sprache ergab sich demnach aus typischen Anforderungen an datengetriebene Business Apps. Dieses Prinzip steht im Gegensatz zu einem *Bottom-Up*-Vorgehen, bei dem ausgehend von den Funktionen, die mobile Plattformen typischerweise zur Verfügung stellen, die Sprachelemente festgelegt würden. Letzterer Gestaltungsansatz zielte auf eine Überdeckung der API mobiler Plattformen. Es bestünde die Gefahr, an den Anforderungen der Sprachanwender vorbei eine Sprache mit niedrigem Abstraktionsniveau zu entwerfen, die den kleinsten gemeinsamen Nenner mobiler Plattformen unterstützt. Unser *Top-Down*-Ansatz hat hingegen den Anspruch, dass

jedes Sprachelement eine spezifische Anforderung der letztendlichen Apps erfüllt. Entwickler sollen mit MD^2-DSL den Problemraum anstelle des Lösungsraums modellieren. Code-Generatoren sind dann für die Transformation in den Lösungsraum verantwortlich.

Diesem Ansatz folgend war unser Ausgangspunkt deshalb eine Erhebung typischer Anforderungen auf Basis von Fachkonzepten und Interviews mit Verantwortlichen. Ergebnis war die folgende Liste: die Sprache soll es App-Entwicklern ermöglichen,

1. Datentypen zu definieren,
2. sowie lokal und serverseitig Datensätze dieser Typen anzulegen, zu lesen, zu aktualisieren und zu löschen, d.h. die typischen CRUD-Operationen auf Instanzen der Typen auszuführen;
3. die Benutzeroberfläche mit verschiedenen Layouts und typischen Steuerelementen zu beschreiben, besonders wichtig sind dabei Registerkarten (Tabs);
4. die Benutzernavigation zwischen den Ansichten zu steuern;
5. Datenbindung und Eingabevalidation zu definieren;
6. auf Benutzerereignisse und Statusänderungen zu reagieren; und
7. gerätespezifische Funktionen wie GPS oder Kamera zu nutzen.

Die Sprachelemente, welche die funktionalen Anforderungen implementieren, und deren Syntaxen werden in den folgenden Unterkapiteln beschrieben. Daneben prägten einige nicht-funktionale Anforderungen die Struktur und Syntax der Sprache. Gefordert waren eine klare Trennung unterschiedlicher Aspekte, Modularisierung, angemessenes Abstraktionsniveau und Verständlichkeit. Trennung und Modularisierung wurden erreicht, indem die Sprache einer MVC-Architektur folgt. Datenmodell (Model), Benutzerschnittstelle (View) und Kontrolllogik (Controller) sind in separaten Dateien zu formulieren, die zusammen das vollständige Modell ergeben. Beziehungen zwischen diesen Teilmodellen folgen dabei dem MVC-Entwurfsmuster. Die Struktur der Sprache ist so flexibel, dass auch diese Teilmodelle modular aufgebaut sind und über mehrere Dateien verteilt werden können. Zum Beispiel könnte jede Registerkarte einer App in einer eigenen Datei definiert werden. Ein modularer Aufbau der App-Beschreibung wird ferner durch Möglichkeiten zur Wiederverwendung, z.B. von Teilen der Benutzeroberfläche, unterstützt.

MD^2-DSL ist vorwiegend deklarativ. Entwickler beschreiben, was die App erreichen soll, aber nicht (mit Ausnahme einiger Aktionen) wie dies algorithmisch ablaufen soll. Deutlich erkennbar ist die deklarative Natur z.B. bei den Sprachelementen für die Benutzeroberfläche. Der Entwickler beschreibt Aussehen und Zusammensetzung der Oberfläche, aber keine Abfolge imperativer Anweisungen zum schrittweisen Aufbau derselben. Der deklarative Sprachstil sorgt in Verbindung mit dem Konvention-über-Konfiguration-Prinzip für eine verständliche und prägnante Sprache mit einem gehobenen Abstraktionsniveau.

MD^2-DSL ist in Xtext [Xte12b] implementiert, einem Framework zur Erstellung textueller Sprachen. Aus einer Sprachbeschreibung in einer attributierten Grammatik mit EBNF-ähnlicher Syntax generiert Xtext einen Parser, die abstrakte Syntax als Klassenmodell und einen Editor für die Eclipse-Umgebung.

```
1 MD2Model ::= PackageDefinition MD2ModelLayer?
2 PackageDefinition ::= 'package' QualifiedName
3 MD2ModelLayer ::= Model | View | Controller
4 QualifiedName ::= ID ('.' ID)*
```

Abbildung 1: Auszug aus der EBNF-Darstellung von MD^2-DSL: Auflösung des Startsymbols

```
1 MD2Model:
2   package=PackageDefinition
3   modelLayer=MD2ModelLayer? ;
4 MD2ModelLayer: Model | View | Controller ;
5 PackageDefinition:
6   'package' pkgName=QualifiedName ;
7 QualifiedName: ID ('.' ID)* ;
```

Abbildung 2: Pendant zu Abbildung 1 in Xtext

Ausschnitte aus der Syntax von MD^2-DSL sind im Folgenden im W3C-Stil der Erweiterten Backus-Naur-Form [W3C04] dargestellt. Alle Nichtterminal-Symbole beginnen mit einem Großbuchstaben. Als vordefinierte Symbole werden `ID`, `INT` und `STRING` verwendet. Diese symbolisieren die Menge aller gültigen Bezeichner, nicht-negativen Zahlen beziehungsweise Zeichenketten.

Abbildung 1 enthält die Regeln zur Auflösung des Startsymbols `MD2Model`, die somit die grundlegende Struktur von MD^2-DSL-Modellen beschreiben. Deutlich wird die generelle Aufteilung in Model, View und Controller (Zeile 3). Jedes Teilmodell beschreibt einen dieser Bereiche. Die entsprechenden Nichtterminal-Symbole werden in den folgenden Unterkapiteln weiter aufgelöst. Das Nichtterminal-Symbol **QualifiedName** wird an den Stellen der Grammatik verwendet, an denen Referenzen zwischen Elementen spezifiziert werden. Xtext ermöglicht für solche Querverweise zusätzlich, auf Ebene der abstrakten Syntax zu beschränken, welcher Typ von Elementen referenziert werden kann. Des Weiteren kann der Gültigkeitsbereich von Elementen feingranular festgelegt werden. Zur Auflösung der Querverweise sieht Xtext nach dem Parsen eine Phase des Linkings vor.

Abbildung 2 enthält die zu Abbildung 1 korrespondierende Implementierung in Xtext, die zusätzlich zur Definition der konkreten Syntax über Zuweisungen (z.B. in Zeile 2) die abstrakte Syntax festlegt. Die vollständige Xtext-Grammatik von MD^2-DSL ist aus Platzgründen unter http://www.wi.uni-muenster.de/pi/forschung/md2/MD2.xtext verfügbar.

Die folgende Darstellung von MD^2-DSL erläutert die Sprache in Ergänzung zu ihrer Grammatik am Beispiel einer vereinfachten Bestell-App, die mit MD^2 implementiert wurde. In ihrer ersten Ansicht kann der Benutzer den Namen eines Produkts eingeben und nach diesem suchen. Das von einem Server übermittelte (zu Illustrationszwecken eindeutige) Ergebnis stellt eine zweite Ansicht dar, in der der Benutzer eine Bestellung aufgeben kann. Zur Vereinfachung genügt dazu die Angabe seiner E-Mail-Adresse. Die Abbildungen 3, 5 und 7 zeigen den nahezu vollständigen MD^2-DSL-Quelltext der App, aufgeteilt nach den Sprachkomponenten Model, View und Controller. Abbildung 9 stellt Screenshots der iOS- und Android-Apps gegenüber, die von MD^2 aus dem Quelltext generiert worden sind.

```
1  package de.md2.bestellung.models
2  entity PRODUKT {
3    name : string
4    preis : integer
5    beschreibung : string(optional)
6  }
7  entity BESTELLUNG {
8    produkt : PRODUKT
9    email : string
10 }
```

Abbildung 3: Datenmodell einer Beispiel-App in MD^2-DSL

```
1 Model ::= ModelElement*
2 ModelElement ::= Entity | Enum
3 Entity ::= 'entity' ID '{' Property* '}'
4 Property ::= ID ':' PropertyType ('(' TypeParam (',' TypeParam)* ')')?
5 PropertyType ::= QualifiedName | 'integer' | 'string' | 'date' | ...
```

Abbildung 4: EBNF-Darstellung der Grammatik von MD^2-DSL: Auszug aus dem Model-Teil

4.2 Datenmodellierung in MD^2-DSL

MD^2-DSLs Komponente zur Datenmodellierung stellt die üblichen Elemente zur Beschreibung von Typen bereit. Sogenannte Entities werden mit dem entsprechenden Schlüsselwort und einem Bezeichner eingeleitet (Abb. 4, Zeile 3). Eingeschlossen in geschweifte Klammern folgt eine Liste von Eigenschaften, bestehend aus Bezeichner und Typangabe. Neben vordefinierten Standardtypen, u. A. für Zeichenketten, ganze Zahlen und Datumsangaben, können Eigenschaften auch auf andere Entities über deren qualifizierten Bezeichner verweisen (Zeile 5). An dieser Stelle sind alle im selben Paket spezifizierten Entities allein über ihren Bezeichner referenzierbar. Die Xtext-Implementierung definiert, dass hier tatsächlich nur Bezeichner von Entities gültig sind. Falls nötig, können Typparameter eine Eigenschaft näher spezifizieren, z.B. als optional oder mehrwertig, aber auch durch komplexere Einschränkungen des zulässigen Wertebereichs.

Zusätzlich zu Entities unterstützt MD^2-DSL Enumerationen. Komplexere Typbeziehungen wie Vererbung werden absichtlich nicht unterstützt, um die Komplexität der Sprache zu reduzieren. Da sich dieser Teil der Sprache vorwiegend bekannter Konzepte bedient, ermöglicht er App-Entwicklern einen einfachen Einstieg.

4.3 Beschreibung der Benutzerschnittstelle mit MD^2-DSL

Die Sprachkomponente für die Benutzerschnittstelle stellt zwei Arten von Anzeigeelementen zur Verfügung: Inhaltselemente und Container (Abb. 6, Zeile 3). Inhaltselemente wie Label, Buttons oder Texteingabefelder werden in Containern gruppiert. Verschiedene Containerarten (*Panes*) repräsentieren unterschiedliche Layouts. Sie können wiederum in anderen Containern geschachtelt werden. Eine besondere Art von Container sind *TabbedPanes* für Benutzeroberflächen mit App-typischer Registernavigation. Die direkten Kind-Container eines TabbedPane bilden dabei die Registerkarten.

```
1  package de.md2.bestellung.views
2  TabbedPane AppFenster {
3    SuchenTab -> Suchen
4    BestellenTab -> Bestellen(tabTitle "Bestellung")
5    InfoTab(tabTitle "Info")
6  }
7  FlowLayoutPane SuchenTab(vertical) {
8    Label sucheLbl { text "Suche nach Produkt" style Gross }
9    TextInput suchFeld { label "Produktname"
10     tooltip "Geben Sie den Namen des Produkts ein, ..."
11   }
12   Button sucheBtn ("Suchen")
13 }
14 FlowLayoutPane BestellenTab(vertical) {
15   Label bestellenLbl { text "Bestellung aufgeben" style Gross }
16   Label info("Bitte geben Sie Ihre E-Mail-Adresse an, ...")
17   AutoGenerator bestellung { contentProvider bestellungProvider }
18   Button bestellenBtn ("Bestellen")
19 }
20 FlowLayoutPane InfoTab { ... }
21 style Gross { fontSize 20 textStyle bold }
```

Abbildung 5: Modell der Benutzerschnittstelle einer Beispiel-App in MD^2-DSL

```
1  View ::= ViewElement*
2  ViewElement ::= ViewGUIElement | Style
3  ViewGUIElement ::= ContentElement | ContainerElement
4  ContentElement ::= Label | TextInput | Button | ...
5  Label ::= 'Label' ID ( '(' STRING ')' | '{' 'text' STRING ('style' ID )? '}' )
6  ContainerElement ::= FlowLayoutPane | GridLayoutPane | TabbedPane | ...
7  FlowLayoutPane ::= 'FlowLayoutPane' ID ('(' FlowParam (',' FlowParam )* ')')?
8                     '{' PaneContent* '}'
9  PaneContent ::= ViewGUIElement | (QualifiedName ('->' ID)?)
```

Abbildung 6: EBNF-Darstellung der Grammatik von MD^2-DSL: Auszug aus dem View-Teil

Um eine modulare Beschreibung der Oberfläche zu ermöglichen, bietet MD^2-DSL die Option, Anzeigeelemente nicht nur direkt zu schachteln, sondern auch anderweitig definierte Elemente zu referenzieren (Zeile 9). Dieses Feature kann wie im Beispiel (Abb. 5) genutzt werden, um einzelne Registerkarten oder Teile der Oberfläche separat zu definieren und die Übersichtlichkeit zu erhöhen. Es ermöglicht ebenso Wiederverwendung. Über ihren qualifizierten Namen referenzierten Anzeigeelementen kann ein neuer Bezeichner zugewiesen werden, mit dem die spezifische Instanz im Controller referenziert werden kann.

Auch die UI-Komponente von MD^2-DSL nutzt allgemein bekannte Konzepte und setzt damit Anforderung 3 auf einfach erlernbare, aber mächtige Weise um. Ein besonderes Element sind sogenannte AutoGenerator-Elemente (Abb. 5, Zeile 17). Diese stehen für eine Standarddarstellung eines Entity-Typs (im Beispiel: BESTELLUNG) mit entsprechenden Labeln und Eingabefeldern. Sie implizieren zudem Datenbindungen zwischen den Inhaltselementen und Objekten eines Datenlieferanten (*ContentProvider*, s.u.) und entledigen den Entwickler, falls gewünscht, manuellen Aufwands. Auch besondere und individuelle Anforderungen an das Design von Apps unterstützt MD^2-DSL, z.B. indem der Stil von Inhaltselementen spezifiziert oder Grafiken eingebunden werden können. Die aus dem GUI-Modell generierte Oberfläche der Beispielapp stellt Abbildung 9 dar.

```
1  package de.md2.bestellung.controllers
2  main {
3    appName "Bestellapp" appVersion "ATPS 2013" modelVersion "1.0"
4    startView AppFenster.Suchen
5    onInitialized init
6  }
7  action CombinedAction init {
8    actions ereignisseRegistrieren validatorenBinden
9  }
10 action CustomAction ereignisseRegistrieren {
11   bind action produktLaden on SuchenTab.sucheBtn.onTouch
12   bind action bestellungAufgeben on BestellenTab.bestellenBtn.onTouch
13 }
14 action CustomAction validatorenBinden {
15   bind validator RegExValidator(regEx "...")
16     on BestellenTab.bestellung[Bestellung.email]
17 }
18 action CustomAction produktLaden {
19   call DataAction(load suchergebnis)
20   call NewObjectAction(bestellungProvider)
21   call AssignObjectAction(use suchergebnis for bestellungProvider.produkt)
22   call GotoViewAction(AppFenster.Bestellen)
23 }
24 action CustomAction bestellungAufgeben {
25   call DataAction(save bestellungProvider)
26 }
27 contentProvider Produkt suchergebnis { providerType backend
28   filter first where Produkt.name equals AppFenster.Suchen->SuchenTab.suchFeld
29 }
30 contentProvider Bestellung bestellungProvider {providerType backend }
31 remoteConnection backend { uri "http://.../de.md2.bestellung.backend/service" }
```

Abbildung 7: Modell der Kontrolllogik einer Beispiel-App in MD^2-DSL

4.4 Beschreibung der Kontrolllogik und des Verhaltens mit MD^2-DSL

Während Model und View die statischen Teile einer App umfassen, steuert der Controller einer MD^2-App deren Verhalten. Die entsprechende MD^2-DSL-Komponente bringt dazu Model und View zusammen. Ein `Main`-Block definiert übergreifende Informationen und spezifiziert den Anfangszustand der GUI sowie beim Start auszuführende Aktionen.

Aktionen sind das zentrale Element in MD^2-DSL zur Definition dynamischer Aspekte. Die mächtigen individuellen Aktionen (*CustomAction*) erlauben es, eine sequentiell auszuführende Liste von Aktionsfragmenten zu definieren. Eine wichtige Art derartiger Fragmente registriert wiederum Aktionen als Behandler für bestimmte Ereignisse (Abb. 8, Zeile 6), wodurch interaktive Benutzerschnittstellen realisiert werden können (Anforderung 6). Neben Interaktionen des Benutzers mit der GUI kann auch auf Veränderungen des globalen Zustandsraums reagiert werden, z.B. auf einen Abbruch der Netzwerkverbindung. Das Sprachkonzept der Bedingungsereignisse erlaubt die Spezifikation komplexer Bedingungen unter Rückgriff auf den Zustand von Model und View, bei deren Erfüllung die für das Ereignis registrierten Aktionen ausgeführt werden. Die hierfür notwendige Sprache für boolesche Ausdrücke wird auch an anderen Stellen in MD^2-DSL verwendet, z.B. für Selektionsausdrücke in Datenabfragen. MD^{2}s dynamische Komponente ist vorwiegend ereignisbasiert, da Aktionen – mit Ausnahme der Startaktion – nur als Reaktion auf Ereignisse ausgeführt werden, an die sie zuvor, zumeist in der Startaktion, gebunden wurden.

Die Aktionen selbst werden als Sequenz definiert und haben einen imperativen Charakter, sind aber auf einem hohen Abstraktionsniveau in das deklarative Umfeld eingebettet. Die Sprache unterstützt keine anderen Kontrollstrukturen.

Die Angabe des Anzeigeelements, an das ein Ereignis gebunden werden soll, erfolgt wie alle Verweise auf ein Anzeigeelement, das im View-Modell definiert wurde, in MD^2-DSL über das Konzept der Anzeigeelement-Referenz (*ViewGUIElementRef*, Abb. 8, Zeile 8 f.). Im einfachsten Fall handelt es sich um den qualifizierten Bezeichner eines Anzeigeelements. Bei wiederverwendeten, referenzierten Anzeigeelementen besteht die Referenz aus zwei Teilen: dem qualifizierten Bezeichner zu der Stelle, an der die Einbindung definiert ist, und, abgetrennt durch einen Pfeil, dem Bezeichner innerhalb des eingebundenen Elements. Auf diese Weise kann ein spezifisches Vorkommen referenziert werden (Abb. 7, Zeile 28). Es ist aber genauso möglich, alle Vorkommen eines Anzeigeelements zu referenzieren, indem nach der ersten Methode der qualifizierte Bezeichner der allgemeinen Definition genutzt wird. Wenn die auf eine dieser Arten definierte Referenz auf einen AutoGenerator verweist, kann zusätzlich in eckigen Klammern eine Eigenschaft des Entity-Typs angegeben werden, für den der Generator definiert wurde (Zeile 16). Die Anzeigeelement-Referenz bezieht sich dann auf das dieser Eigenschaft entsprechende implizit generierte Feld.

Neben dem erwähnten Fragment zur Ereignisbindung stellt MD^2-DSL eine Vielzahl weiterer Aktionsfragmente bereit. Sie ermöglichen beispielsweise, Inhaltselemente mit Validatoren und Datenbindungen zu versehen (siehe unten), oder gerätespezifische Aktionen (GPS-Lokation, Anforderung 7) auszuführen. Andere Fragmente steuern die Navigation innerhalb der grafischen Oberfläche der App und können zwischen Ansichten wechseln. Die CRUD-Operationen werden ebenfalls durch mehrere Aktionsfragmente unterstützt.

Als Sprache für die Beschreibung datenzentrierter Business Apps bietet MD^2-DSL Elemente zur Verknüpfung der Apps mit lokalen und entfernten Datenquellen. Beide werden über Datenlieferanten (*ContentProvider*) angebunden. Ein Lieferant verweist auf einen Datenspeicher, bei dem es sich entweder um eine Datenbank auf dem Gerät oder um einen Server handelt, der über eine vorgegebene API Zugriff bietet. Jeder Datenlieferant verweist auf einen spezifischen Ausschnitt der Datenquelle, der über den Typ der Daten und einen Filter definiert wird. Im Beispiel verweist der Lieferant `suchergebnis` (Zeile 27 ff.) auf das Produkt, dessen Namen der Eingabe im Suchfeld entspricht. Der Typ des Datenlieferanten wird so angegeben, wie er auch für Eigenschaften in der Datenkomponente von MD^2-DSL definiert ist, d.h. er verweist entweder auf einen vordefinierten Datentyp oder einen im Datenmodell deklarierten Entitytyp; zudem kann er mehrwertig sein. Ein Filter entspricht einer Abfrage, die Objekte selektiert, die der angegebenen Bedingung entsprechend. Im einfachsten Fall entspricht die Bedingung einem Vergleich einer Eigenschaft des Entitytyps, angegeben als qualifizierter Bezeichner, mit einem Anzeigeelement, auf das über die beschriebene Anzeigeelement-Referenz verwiesen wird (Abb. 8, Zeile 13).

Verschiedene Aktionsfragmente von MD^2-DSL operieren auf Datenlieferanten und setzen zusammen mit diesen Anforderung 2 um. Eine *DataAction* spezifiziert, dass auf dem durch den Datenlieferanten bezeichneten Ausschnitt eine der CRUD-Operationen Speichern, Laden oder Löschen ausgeführt wird. Eine neue Initialisierung ist mit einer *NewObjectAction* möglich, während Assoziationen zwischen den Entities zweier nicht-mehrwertiger

```
1  Controller ::= ControllerElement*
2  ControllerElement ::= Main | Action | ContentProvider | Workflow | ...
3  Action ::= 'action' (CustomAction | CombinedAction)
4  CustomAction ::= 'CustomAction' ID '{' CodeFragment* '}'
5  CodeFragment ::= EventBinding | ValidatorBinding | ActionCall | Mapping | ...
6  EventBinding ::= 'bind action' QualifiedName* 'on' (GUIEvent | GlobalEvent)
7  GUIEvent ::= ViewGUIElementRef '.' EventType
8  ViewGUIElementRef ::= QualifiedName ('->' QualifiedName)*
9                        ('[' QualifiedName ']')?
10 ContentProvider ::= 'contentProvider' PropertyType '[]'? ID
11                     '{' ProviderType Filter? '}'
12 Filter ::= 'filter' FilterType ('where' WhereClause)?
13 WhereClause /*stark verkürzt*/ ::= QualifiedName 'equals' ViewGUIElementRef
```

Abbildung 8: EBNF-Darstellung der Grammatik von MD^2-DSL: Auszug aus dem Controller-Teil

Datenlieferanten über *AssignObjectAction* gepflegt werden. Dabei wird die aktuelle Entity eines Lieferanten einer Eigenschaft der Entity eines zweiten Lieferanten zugewiesen. Die Bestell-App nutzt das geladene Suchergebnis als Produkt für eine neu initialisierte Bestellung (Abb. 7, Zeilen 19–21).

Datenbindungen zwischen Datenlieferanten und Inhaltselementen sind als Aktionsfragmente realisiert und können als solche dynamisch zugewiesen werden. Eine Datenbindung verknüpft ein Anzeigeelement mit einer Eigenschaft eines Datenobjekts, das von einem Datenlieferanten verwaltet wird. Sie stellt dauerhaft die Konsistenz zwischen Daten und GUI in beide Richtungen sicher (Anforderung 5). Im Fall eines Eingabefelds impliziert eine Datenbindung zudem einen Validator, der vom Datentyp und zusätzlich spezifizierten Einschränkungen abgeleitet wird. Das Überschreiben und Ergänzen impliziter Validatoren ist über entsprechende Aktionsfragmente möglich. Eingaben können z.B. darauf überprüft werden, ob sie vorhanden sind, ob sie konform zu einem Datentyp sind oder ob sie einem regulären Ausdruck entsprechen. Bei AutoGeneratoren werden sowohl Datenbindungen als auch Validatoren zunächst automatisch abgeleitet, was in der Beispielapp genügt.

Ein fortgeschrittenes Sprachelement von MD^2-DSL ist die Möglichkeit, Navigationspfade durch die App im Sinne von *Workflows* zu spezifizieren. Sie sind insbesondere in Registerkarten-basierten Benutzeroberflächen wichtig, um den Nutzer einer App angesichts vergleichsweise vielfältiger Optionen bei der Navigation zu unterstützen. Aus Platzgründen kann das Konzept hier nur angerissen werden und wird nicht im Beispiel genutzt. Ein Workflow besteht aus mehreren Schritten. Jedem Workflow-Schritt ist ein Container der GUI zugewiesen, der angezeigt wird, falls der Schritt aktiv ist. Für einen Schritt definierte Bedingungen – z.B. vollständig ausgefüllte Eingabefelder – beeinflussen, wann ein anderer Schritt aktiv werden kann. Verschiedene Aktionsfragmente ermöglichen es in Verbindung mit Workflows, die Navigationspfade einer App zu weiter spezifizieren.

5 Generierung von iOS- und Android-Apps

Die Spezifikation von MD^2-DSL, der DSL von MD^2, entspricht auch dem Funktionsumfang des Frameworks insgesamt, da die Menge gültiger Modelle in MD^2-DSL die Menge möglicher Eingaben des Code-Generators darstellt. Das vorstehende Kapitel beschreibt

neben den zur Verfügung stehenden Sprachelementen auch deren Semantik, soweit es der beschränkte Platz erlaubt. Dieser Abschnitt geht kurz auf die Implementierung der drei Code-Generatoren ein (Details in [HMK13]). Wie eingangs beschrieben, werden vollständige und lauffähige Android- und iOS-Apps sowie eine JavaEE-Anwendung generiert, welche die Serverschnittstelle von Datenlieferanten implementiert.

Jeder Code-Generator ist in Xtend [Xte12a] implementiert, einer Java-ähnlichen Programmiersprache mit für Codegeneratoren hilfreicher Zusatzfunktionalität wie Template-Ausdrücken. Wie Abbildungen 10 und 11 am Beispiel der Generierung von zu Entities korrespondierendem Quelltext demonstrieren, sind die Generatoren hinreichend unterschiedlich, um voneinander unabhängig implementiert zu sein. Abfragen auf der abstrakten Syntax des Eingabemodells werden aber von allen Generatoren genutzt. Zudem reichert ein vorverarbeitender Schritt das Modell mit zusätzlichen Informationen an und modifiziert es, um die Codegenerierung zu vereinfachen. So werden beispielsweise Auto-Generatoren durch explizite Anzeigeelemente, Datenbindungen und Validatoren ersetzt.

Hauptaufgabe der Generatoren ist es, die deklarativen Sprachkonzepte auf den jeweiligen Zielplattformen zu explizieren. Eine Zielplattform ist nicht nur durch die Programmiersprache bestimmt, in der generierter Code verfasst ist, sondern auch durch die zur Verfügung stehenden Bibliotheken und statischen, von MD^2 mitgelieferten Inhalt. Die Generatoren folgen den Richtlinien der jeweiligen mobilen Plattform und erzeugen z.B. XML-Dokumente für das Layout von Android-Apps. Die Generatoren nutzen jeweils plattformspezifische Konzepte, um die abstrakten Sprachkonzepte umzusetzen. Wo möglich, greifen sie auf typische Elemente der Plattform zurück, um ein natives Aussehen zu erreichen. Fehlt auf einer Plattform ein Pendant eines bestimmten Konzepts, setzt der Generator dieses aus bestehenden Elementen zusammen, z.B. wird das Grid-basierte Layout auf Android für den App-Nutzer transparent durch ein TableLayout emuliert.

6 Verwandte Arbeiten und Diskussion

MD^2 lässt sich von anderen Ansätzen zur plattformübergreifenden App-Entwicklung abgrenzen. Hierzu gehören: mobile Web-Apps, hybride Apps, Laufzeitumgebungen und generative Ansätze. *Webapps* verwenden HTML, CSS und JavaScript und lassen sich in einem Browser aufrufen. Sie können auf gerätespezifische Features nicht bzw. nur eingeschränkt im Rahmen von HTML 5 zugreifen. Weiterhin vermitteln sie das Bediengefühl einer Webseite und erreichen kein plattformspezifisches Look & Feel. Letzteres ist aber sehr wichtig, wie wir von unseren Praxispartnern lernen konnten. *Hybride Apps* wie Apache Cordova [Apa12] (ehemals PhoneGap) verpacken eine native Komponente in eine Webseite und können so auch gerätespezifische Features verfügbar machen. Aber auch diese erreichen kein plattformspezifisches Look & Feel. MD^2 erreicht dieses plattformspezifische Look & Feel trivialerweise, da jeweils plattformspezifischer Code generiert wird.

Ansätze wie Appcelerator Titanium [App12a], die eine separate Laufzeitumgebung verwenden und hiermit plattformübergreifenden Skript-Code interpretieren, können im Prinzip ein plattformspezifisches Look & Feel erreichen. Allerdings führt die zusätzliche In-

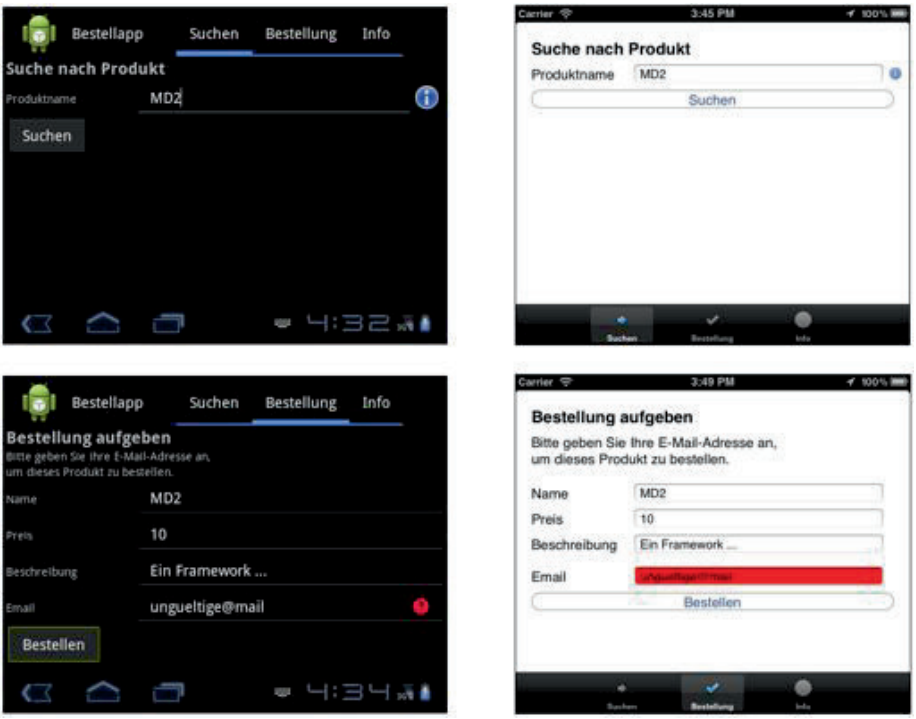


Abbildung 9: Screenshots der Bestell-App für iOS und Android (zu Illustrationszwecken modifiziert)

```
1 public class «entity.name» implements Entity {
2   private int __internalId;
3   «FOR attribute : entity.attributes»
4     private «attributeTypeName(entity, attribute)» «attribute.name»;
5   «ENDFOR»
6 }
```

Abbildung 10: Template zur Generierung von Java-Quelltext für Entities in Android (Auszug)

```
1 @interface «entity.name.toFirstUpper»Entity : DataTransferObject
2
3 «FOR attribute : entity.attributes»
4   @property (retain) «attributeTypeName(entity, attribute)»* «attribute.name»;
5 «ENDFOR»
```

Abbildung 11: Template zur Generierung von Objective-C-Quelltext für Entities in iOS (Auszug)

terpretation zu erheblichen und manchmal prohibitiven Laufzeiteinbußen.

Neben modellgetriebenen Ansätzen gibt es auch weitere generative Ansätze zur App-Entwicklung. Die Cross-Compiler XMLVM [XML12] und J2ObjC [J2O12] beispielsweise transformieren Java- in Objective-C-Code (für iOS), berücksichtigen aber keine gerätespezifische Funktionalität und sind auf iOS begrenzt. Alternative modellgetriebe-

ne Ansätze wie applause [app12b] und AXIOM [JJ12] generieren ebenso wie MD^2 Apps aus einer DSL. Applause ist auf Informationsdarstellung beschränkt. Im Fall von AXIOM wurden die Sprachelemente bottom-up aus den Geräte-Features abgeleitet. Die betreffende DSL arbeitet daher auf einem niedrigen Abstraktionsniveau; zudem ist die Transformation nicht vollständig automatisiert. Wie alle zuvor erwähnten Ansätze profitieren sie daher im Gegensatz zu MD^2 nicht von Besonderheiten der Anwendungsdomäne Business Apps.

MD^2 erzeugt dagegen aus sehr kompakten DSL-Modellen effizienten, nativen Code für jede Zielplattform. Die DSL-Features wurden nicht aus den Gerätemerkmalen sondern aus den Anforderungen der Anwendungsdomäne abgeleitet. Ein Tarifrechner, der in Zusammenarbeit mit einem Praxispartner aus der Versicherungsbranche entwickelt wurde, konnte beispielsweise in 709 DSL-Zeilen formuliert werden, die in 10110 Zeilen Java und 2263 Zeilen XML für Android bzw. 3270 Zeilen Objective-C und 64 Zeilen XML für iOS transformiert wurden. Dies verdeutlicht den Produktivitätsgewinn durch die deutlich höhere Abstraktion. Weiterhin mussten die 709 Zeilen natürlich nur einmal erstellt werden und konnten dann automatisch in Code für jede Plattform transformiert werden.

Unser Ansatz hat weiterhin eine gewisse Ähnlichkeit zu dem Task-orientierten Toolkit iTasks [MPA11], mit dem sich Workflows mit GUI-Elementen erstellen lassen. iTasks verwendet keinen Modell-getriebenen Ansatz sondern monadische Kombinatoren höherer Ordnung ist damit wesentlich komplexer als MD^2-DSL. Außerdem ist es nicht auf mobile Endgeräte zugeschnitten und bietet insofern auch keine Unterstützung gerätespezifischer Funktionen auf verschiedenen Plattformen.

MD^2 ist nicht für jede App geeignet, sondern gezielt auf datengetriebene Business Apps zugeschnitten, bei denen die Benutzerschnittstelle im Wesentlichen aus Textfeldern, Labels, Buttons, Menüs und Listen besteht. MD^2 ist insbesondere ungeeignet für Apps, bei denen im Gerät komplexe Algorithmen ausgeführt werden müssen, wie dies zum Beispiel bei Spielen oder Multimedia-Anwendungen der Fall ist. MD^2 unterstützt nur die gerätespezifischen Features (wie z.B. GPS), die für Business Apps benötigt werden. Hinsichtlich der zur Verfügung gestellten Kontrollstrukturen ist MD^2 bewusst limitiert. Verzweigungen sind nur auf Grundlage der festgestellten Ereignisse möglich. Schleifen lassen sich in eingeschränkter Form durch Rückkehr zu einem zuvor besuchten Bildschirm realisieren. Komplexere Algorithmen werden bei Business Apps typischerweise auf der Server-Seite ausgeführt. Bei allen bisher betrachteten Business Apps (neben einem Tarifrechner für ein Versicherung u.a. eine Bestell-App für einen E-Shop und eine Bibliotheks-App) reichten die verfügbaren Kontrollstrukturen aus. Um die DSL kompakt zu halten, haben wir daher auf Weitere verzichtet. Sollte sich zukünftig ein Bedarf an weiteren Kontrollstrukturen abzeichnen, so können diese gegebenenfalls später ergänzt werden. Die Einbeziehung eines Servers ist typisch, aber nicht zwingend erforderlich. Auch Apps, bei denen Daten lokal auf dem Gerät gespeichert und durch simple Workflows manipuliert werden, sind realisierbar.

7 Zusammenfassung und Ausblick

Wir haben die textuelle domänenspezifische Sprache MD^2-DSL vorgestellt, die sich zur plattformübergreifenden Entwicklung datengetriebener Business Apps für mobile Endgeräte eignet. MD^2-DSL-Dokumente werden unter Verwendung von Xtext und Xtend automatisch in Code für die jeweilige Zielplattform übersetzt. Zur Zeit werden die Plattformen iOS/Objective-C und Android/Java unterstützt. Weitere Plattformen werden später ergänzt. Im Gegensatz zu anderen DSLs zur App-Entwicklung wurde MD^2-DSL nicht bottom-up aus den Features mobiler Endgeräte sondern top-down aus den Anforderungen an Business Apps konzipiert und erreicht damit ein gegenüber anderen Ansätzen deutlich höheres Abstraktionsniveau. Insbesondere unterstützt MD^2-DSL die bekannte Model-View-Controller-Architektur. Die View-Komponente wird deklarativ durch Schachtelung von Containern und Platzierung von Bedienelementen wie Texteingabefeldern und Knöpfen beschrieben. Das Model besteht im Wesentlichen aus einer ebenfalls deklarativen Beschreibung der benötigten Entitäten. Die Controller-Komponente legt schließlich fest, wie die relevanten Ereignisse behandelt werden sollen. Weiterhin können Validatoren zur Überprüfung von Eingaben angegeben werden. Einen erheblichen Teil einer typischen Business App machen simple CRUD-Operationen auf den Entitäten aus. Eine zugehörige Benutzerschnittstelle hierfür kann automatisch auf Basis von MD^2-DSLs Datenlieferanten und Autogeneratoren generiert werden.

Manche anderen Ansätze zur plattformübergreifenden App-Entwicklung verlieren Effizienz durch das Einschieben einer zusätzlichen Skript-Code-Interpretationsschicht. Bei unserem Ansatz ist dies nicht der Fall. Es wird direkt plattformspezifischer Code erzeugt, wie man ihn in etwa auch von Hand schreiben würde. Dieser plattformspezifische Code garantiert insbesondere auch das Look & Feel der jeweiligen Zielplattform. Diese sehr wichtige Eigenschaft wird von den meisten anderen Ansätzen zur plattformübergreifenden App-Entwicklung nicht erreicht. Wie sich an einigen Beispielanwendungen mit einem Praxispartner aus der Versicherungsbranche zeigte, erlaubt MD^2-DSL eine sehr kompakte Formulierung von Business Apps und damit eine hohe Produktivität. Mehraufwand für die Übertragung einer App auf andere Plattformen entfällt.

Literatur

[Apa12] Apache Cordova, 2012. http://incubator.apache.org/cordova/.

[App12a] Appcelerator, 2012. http://www.appcelerator.com/.

[app12b] applause, 2012. https://github.com/applause/.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[Gar12] Gartner Press Release, 2012. http://www.gartner.com/it/page.jsp?id=1622614.

[HMK13] Henning Heitkötter, Tim A. Majchrzak und Herbert Kuchen. Cross-Platform Model-Driven Development of Mobile Applications with MD2. In *Proc. of the 2013 ACM Symp. on Applied Computing (SAC)*. ACM, 2013.

[J2O12] J2ObjC, 2012. https://code.google.com/p/j2objc/.

[JJ12] Xiaoping Jia und Christopher Jones. AXIOM: A Model-driven Approach to Cross-platform Application Development. In *Proc. 7th ICSOFT*, 2012.

[MPA11] Steffen Michels, Rinus Plasmeijer und Peter Achten. iTask as a New Paradigm for Building GUI Applications. In *Proceedings of 22nd IFL, LNCS 6647*, Seiten 153–168, 2011.

[SV06] Thomas Stahl und M. Völter. *Model-driven software development*. John Wiley & Sons New York, 2006.

[W3C04] Extensible Markup Language (XML) 1.0. Notation. Bericht, W3C, 2004. http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation.

[XML12] XMLVM, 2012. http://www.xmlvm.org/.

[Xte12a] Xtend, 2012. http://www.eclipse.org/xtend/.

[Xte12b] Xtext, 2012. http://www.eclipse.org/Xtext/.

How useful are existing monitoring languages for securing Android apps?

Steven Arzt[1], Kevin Falzon[1], Andreas Follner[1], Siegfried Rasthofer[1], Eric Bodden[1] and Volker Stolz[2]

[1] Secure Software Engineering Group, EC SPRIDE,
Technische Universität Darmstadt, Germany
{firstname.lastname}@ec-spride.de
[2] Precise Modelling and Analysis Group, University of Oslo, Norway
stolz@ifi.uio.no

Abstract: The Android operating system is currently dominating the mobile device market in terms of penetration and growth rate. An important contributor to its success are a wealth of cheap and easy-to-install mobile applications, known as *apps*. Today, installing *untrusted* apps is the norm, though this comes with risks: malware is ubiquitous and can easily leak confidential and sensitive data.

In this work, we investigate the extent to which we can specify complex information flow properties using existing specification languages for runtime monitoring, with the goal to encapsulate potentially harmful apps and prevent private data from leaking. By modelling a set of representative, Android-specific security policies with Tracematches, JavaMOP, Dataflow Pointcuts and PQL, we are able to identify policy-language features that are crucial for effectively defining runtime-enforceable Android security properties.

Our evaluation demonstrates that while certain property languages suit our purposes better than others, they all lack essential features that would, if present, allow users to provide effective security guarantees about apps. We discuss those shortcomings and propose several possible mechanisms to overcome them.

1 Introduction

According to a recent study [Cor12], Android now has about 75% market share in the mobile-phone market, with a 91.5% growth rate over the past year. With Android phones being ubiquitous, they become a worthwhile target for security and privacy violations. Attacks range from broad data collection for the purpose of targeted advertisement, to targeted attacks, such as the case of industrial espionage. Attacks are most likely to be motivated primarily by a social element: a significant number of mobile-phone owners uses their device both for private and work-related communication [Bit12]. Furthermore, the vast majority of users install apps containing code whose trustworthiness they cannot judge and which they cannot effectively control.

These problems are well known, and indeed the Android platform does implement state-of-the-practice measures to impede attacks. The Android platform is built as a stack, with

various layers running on top of each other [And12].

The lower levels consist of an embedded Linux system and its libraries, with Android applications residing at the very top. Users typically acquire these applications through various channels (e.g., the Google Play marketplace[1]). The underlying embedded Linux system provides the enforcement mechanisms common to the Linux kernel, such as a user-based permission model, process isolation and secure inter-process communication. By default, an application is not allowed to directly interact with other applications, operating system processes, or a user's private data [Goo12]. The latter includes, for example, access to the contacts list. Access to such private data is regulated by Android via a *permission-based* security model, where applications have to statically declare the permissions they require in order to access security-sensitive API functions. An application may only be installed following the user's informed consent, yet users currently have little control over the installation process, as they must either grant all of the permissions that an app demands, or else forego installation.

Popular Android extensions such as AppGuard [BGH+12] mitigate this problem by instrumenting apps with dynamic permission checks at installation time. Through this mechanism, users can revoke permissions they had initially granted at installation time. While a definite improvement, permission revocation is not a complete solution. Part of the problem originates from the coarse-grained nature of Android permissions [NKZ10, ZZJF11]. For instance, an app may require internet and phone-book access permissions to function correctly, in which case revoking either would not be an option. Nevertheless, one may wish to forbid the app from transmitting phone book entries over the internet. Such fine-grained restrictions on information flow are not possible with Android's existing permission-based security model.

One possible solution to the problem are specialized *runtime monitoring approaches* for information flow properties. In the past, researchers have proposed several different monitor specification languages for specifying policies that, through a runtime monitoring tool, are automatically enforced as the monitored program executes. These tools typically instrument the program artifact directly rather than its source code. Given a policy definition and a potentially unsafe or insecure program, a specialized compiler or weaver instruments the program with security checks which ensure that every program execution that violates the policy is detected. Developers can use this information, e.g. for logging violations or defining countermeasures to abort or gracefully handle problematic executions.

Yet, how well do these specification languages address the problems at hand? Can any of the currently available languages be used to effectively specify and enforce security and privacy policies that are of practical interest in the context of Android? In this work, we aim to answer this question by evaluating four different monitor specification languages. Our work focuses on languages, rather than tools, because we believe that approaches for information-flow analysis need to be customizable, and that users of those tools should be able to prove security guarantees over monitored apps. Languages have the potential to allow for a level of abstraction that permits such proofs.

We specifically exclude pure information-flow analysis tools that do not offer a policy lan-

[1] Available at `https://play.google.com/` (December 2012)

guage, for instance TaintDroid [EGgC+10]. While such tools present important backend technology that allows for efficient analysis, in this paper we aim to answer important questions on specification languages, i.e., on frontend technology. With respect to tooling, TaintDroid also differs from the approaches studied here in that it requires modification to the Android runtime, which in an end-user scenario is undesired. All approaches presented here work by instrumenting the app under test; the instrumented app can then execute in a standard execution environment.

We have formulated different Android code snippets that include information-flow properties and tried to enforce security properties of them using the finite-state monitoring languages Tracematches [AAC+05] and JavaMOP [CR07], the AspectJ language extension Dataflow Pointcuts [MK03, ABB+09] and the program query language PQL [MLL05]. We found that some of the languages are effective at securing well-structured programs against input-driven attacks such as SQL injections, a concern typically reserved for server applications. However, in our scenario we must assume that it is the app, not the input, that is malicious, and hence we cannot expect programs to be well-structured, as they can be arbitrarily obfuscated. None of the existing approaches allow security experts to define policies in a way that would allow detecting violations in such a setting. We discuss this problem in detail and present a range of possible solutions that designers of policy languages can choose from to mitigate the problem.

To summarize, this work presents the following original contributions:

- an investigation of the suitability of Tracematches, JavaMOP, Dataflow Pointcuts and PQL for the monitoring of security policies,
- a discussion of the reasons why these languages fail to provide practically enforceable security guarantees, and
- a discussion of possible language-design options that could mitigate the problem covering the areas of recursion and reuse, variable and member access, custom monitor states, global and persistent state, for implicit information flow detection, handling of primitive data types and native code, placement of sanitizers, "proceed" instructions and specification language typing.

2 Example Properties

Recently, several works [ZJ12, EOMC11, EGgC+10, GCEC12, FHM+12, KB12] have investigated the detection of different forms of vulnerabilities in Android apps. A principal concern is the leakage of sensitive information, including the IMEI, IMSI[2] and location information. In addition, there are malicious applications that harm the user financially, for instance by sending premium-rate SMS/MMS messages [EOMC11], or eavesdropping on online banking transactions, stealing the mTAN[3] and withdrawing money from the

[2] International Mobile Equipment/Subscriber Identity

[3] One-time password sent to a user's phone by an online banking service to authorize a transaction.

victim's account [KB12]. Furthermore, Fahl et al. [FHM+12] found various forms of SSL/TLS misuse in Android applications.

The most prevalent vulnerabilities in the mobile scenario are related to information-flow properties, such as sending sensitive information to a specific target. Therefore, we created three different kinds of malicious code snippets (pseudocode) that could be implemented in real Android applications. These snippets will be used as a basis for our evaluation to demonstrate the respective strengths and weaknesses of the different property languages. Listing 1 shows an example of an information-flow property. The sensitive information IMEI together with the location is read from the device and is propagated through the code until it arrives at the "sendTextMessage" sink. At this point the sensitive information is leaked because it leaves the device in form of an SMS message to a specific phone number. This leads to an explicit information-flow violation in the present example. However, Listing 2 shows almost the same example (excluding the location information), but with the difference of an implicit flow. The example converts the IMEI into a bit stream and iterates over the bits. There exists an implicit flow in line 8 for the string argument "0" that is only sent to the specific target under the condition that the bit is 0. Line 10 performs the corresponding action in case the bit is 1, sending the string "1".

```
1  String imei =
     TelephonyManager.getDeviceID();
2
3  double latitude =
     LocationManager.getLatitude();
4
5  String message = "IMEI: " + imei;
6  message += "LOCATION: " +
     latitude;
7
8  sendTextMessage("+44 020 7321
     0905", message);
9
10 ...
```

Listing 1: Violation of an information-flow property (explicit flow)

```
1  String imei =
     TelephonyManager.getDeviceID();
2
3  int[] imeiAsBitStream =
     imei.toBitStream();
4
5  for(int bit : imeiAsBitStream)
6    if(bit == 0)
7      sendTextMessage("+44 020 7321
       0905", "0");
8    else
9      sendTextMessage("+44 020 7321
       0905", "1");
10 ...
```

Listing 2: Violation of an information-flow property (implicit flow)

The example in Listing 3 shows a code snippet that iterates through every phone number in the mobile device's contact list and sends a spam message ("I love you!") to these numbers. In contrast with the previous examples, countermeasures for this code will be more concerned with the frequency of transmission of messages, rather than information leaking into messages. For instance, the policy *"allow at most 3 text messages to be sent from a specific app per day"* would require some kind of counting mechanism in the monitoring language.

```
1  String[] contactPhoneNumbers = getAllPhoneNumbersOfContacts();
2
3  for(String number : contactPhoneNumbers)
4    sendTextMessage(number, "I love you!");
5  ...
```

Listing 3: Spam message to all contacts

3 Tracematches

Tracematches [AAC+05] are an extension to the aspect-oriented programming language AspectJ [KHH+01]. The extension provides users with the ability to define runtime monitors that can match the program's dynamic execution trace against a regular expression of events. Tracematches support free variables in those events, which allows users to relate events to one another on a consistent group of objects. This makes Tracematches an ideal specification language for defining properties in cases where each such group of objects, as well as the state that they share, is finite.

Listing 4 details a compact Tracematch designed to detect and prevent the SMS-spamming behaviour described in Listing 3. The property is triggered whenever two consecutive SMS messages are sent with no intervening user interaction. Security events of interest are defined using AspectJ pointcuts, which map points in the program to symbolic event names. In this case, a pointcut for the SMS sending method is specified, along with a pointcut `userInteraction` defined elsewhere.

The sequence of interest is defined as a regular expression, where events are received and matched against the defined expression. The expression `send_sms send_sms` matches a trace "`send_sms send_sms`" but not "`send_sms user_input send_sms`". Therefore, the error handler would trigger exactly if two consecutive SMS messages are sent without an intervening user action. When the handler triggers, it replaces the call to `sendTextMessage`.

Tracematches are not suitable, however, for defining information-flow properties. This is due to the fact that information flow can propagate through an arbitrary number of variables and objects. While it is possible to define a Tracematch that can detect insecure information flows for any particular code example, one often finds that such properties do not scale very well, and can be very fragile. For example, consider Listing 5, which describes a Tracematch designed to detect the violation of the explicit flow property presented in Listing 1. In Java, string concatenation via the + operator is implemented as a series of append operations on a StringBuilder structure. As a result, the Tracematch must attempt to follow the sensitive data elements as they traverse multiple structures. Consequently, even a seemingly innocuous change of concatenation operators would foil detection. Similarly, detection would be avoided were one to rearrange the sequence in which the concatenation operators take place, or by introducing additional intermediate steps, which may also re-encode the sensitive information, making it harder to keep track of propagations. It is also not possible to generalize a Tracematch such that it would track flows of arbitrary length: any given Tracematch can only reason about a fixed number of values, but information flows can involve an arbitrary number of objects.

```
1 tracematch() {
2     sym user_input before: userInteraction();
3     sym send_sms   around: call(* SmsSession.sendTextMessage(..));
4
5     send_sms send_sms {
6         System.err.println("Sms spam detected! Sending aborted."); }
7 }
```

Listing 4: Tracematch detecting SMS spam

```
tracematch(String imei, double lat, Object msg1, Object msg2, Object msg2_sb,
     Object merged, String msg1_s, String msg2_s, String merged_s) {
  sym ret_imei  after returning(imei):    call(* TelephonyManager.getDeviceID());
  sym ret_lat   after returning(lat):     call(* LocationManager.getLatitude());
  // Trigger when appending sensitive data to a string
  sym appendI   after returning(msg1):    call(* *.append(..)) && args(imei);
  sym appendL   after returning(msg2):    call(* *.append(..)) && args(lat);
  // Translating from internal representation to strings ("message" is a String)
  sym appendI_s after returning(msg1_s):  call(* *.toString()) && target(msg1);
  sym appendL_s after returning(msg2_s):  call(* *.toString()) && target(msg2);
  sym merge_s   after returning(merged_s):call(* *.toString()) && target(merged);
  // Trigger on conversion from String to internal representation
  sym appendL_sb  after returning(msg2_sb):call(*.new(..)) && args(msg1_s);
  // Sensitive data (IMEI and Latitude) have been merged into a single string
  sym merge_sb  after returning(merged):  call(* *.append(..)) && target(msg2_sb)
     && args(msg2_s);
  // Sensitive data leaked via SMS
  sym sendSms   around: call(* SmsSession.send*(..)) && args(*, merged_s);

  // The regular expression
  ret_imei ret_lat appendI appendI_s+ appendL_sb appendL appendL_s+ merge_sb
     merge_s sendSms {
    proceed(sanitize(merged_s)); //send a sanitized version of the SMS
  }
}
```

Listing 5: Tracematch detecting explicit information flow of secret strings

4 JavaMOP

JavaMOP [CR07] is designed to monitor properties defined in a range of different temporal logics. Listing 6 is a reformulation of the security property for detecting SMS spamming. Similar to the Tracematch property described in Listing 4, this property identifies a set of methods related to user interaction, along with the method for transmitting a message. In this example, the property has been expressed as a *Linear-Temporal Logic* (LTL) formula, which states that a `send_sms` event should be followed by a `user_input`.

Listing 7 describes an approach to performing taint tracking in the context of the problem defined in Listing 1. The property definition is less fragile than the Tracematch definition. In particular, it completely abstracts from the order in which events occur. Also, by using a custom data structure `taintedStrings` with membership queries, the specification can define the property recursively: a string is tainted if it is returned from a source or it is built from a tainted string. As HashSets use equality, and not identities, this solution may flag (string-representations of) values as tainted *regardless of their source*.

```
SmsSpam() {
  event user_input before() : userInteraction() { }
  event send_sms before() :  call(* SmsSession.sendTextMessage(..))  { }
  // If SMS sent, next event must be an interaction
  ltl: [](send_sms => o user_input)

  @violation { System.err.println("Sms spam detected!"); }
}
```

Listing 6: JavaMOP (LTL plugin) detecting SMS spam

```
ExplicitSpec() {
  Set <String> taintedStrings = new HashSet<String>();

  void taint(String s) { taintedStrings.add(s); }

  boolean isTainted(String s) { return taintedStrings.contains(s); }

  event retImei after() returning (String imei):
    call(* TelephonyManager.getDeviceID())  { taint(imei); }

  event retLat  after() returning (double lat):
    call(* LocationManager.getLatitude())  { taint(new Double(lat).toString()); }

  event propagate_strings after(StringBuilder sb, String s):
    (call(* StringBuilder.append(String)) || call (StringBuilder.new(String)))
     && target(sb) && args(s) {
      if (isTainted(s)) taint(sb.toString());
  }

  event propagate_doubles after(StringBuilder sb, double d):
    (call(* StringBuilder.append(double )) || call (StringBuilder.new(double)))
     && target(sb) && args(d) {
      String s = new Double(d).toString();
      if(isTainted(s)) taint(sb.toString());
    }

  event sink before(String s):
    call (* sendTextMessage(String, String)) && args(t, s) {
      if (isTainted(s))
        System.err.println("Sensitive information (" + s + ") is sent to:" + t);
    }
}
```

Listing 7: Taint tracking using events in JavaMOP

While JavaMOP improves on the Tracematch specification, it is interesting to note that Listing 7 basically uses no JavaMOP features any longer: the same monitor could just as well have been written in plain AspectJ.

5 Dataflow Pointcuts

Alhadidi et al. [ABB+09] have proposed a formal framework for the dataflow pointcut, an original contribution by Masuhara and Kawauchi [MK03]. The dataflow pointcut describes where aspects should be applied based on the origins of data. The suggested use case is the detection of input-validation vulnerabilities, in which case a sanitizer, being the crosscutting concern, is applied.

```
pointcut sendIMEI(String o) :
  call(SmsSession.sendTextMessage(String)) && args(o)
  && dflow[o,i](call(String TelephonyManager.getDeviceID())
    && returns(i));
```

Listing 8: Dataflow pointcut for a sanitization task

A dataflow pointcut for a sanitization task could be defined as in Listing 8. The second line matches calls to the `sendTextMessage` method and binds the parameter string to variable `o`.

The `dflow` pointcut is used to limit the join points to those whose parameter string was filled with the return value of `getDeviceID` at a previous join point. Masuhara and Kawauchi also provide an additional declaration form which makes it possible to specify explicit propagation of dataflow through external program parts. This proves useful when using third party libraries, native code or in any other situation where analyzable code is not available. The example from [MK03] demonstrates the syntax:

```
1 aspect PropagateOverEncryption {
2   declare propagate: call(byte[] Cipher.update(byte[]))
3     && args(in) && returns(out) from in to out;
4 }
```

Here, the system will assume that the return value from `Cipher.update` originates from its argument. As a result, if a string matches the dataflow pointcut, the encrypted string also matches the dataflow pointcut. This idea of explicit data flow propagation has not been adapted by Alhadidi et al. in their formal framework [ABB+09].

The framework proposed by Alhadidi et al. uses dataflow tags which discriminate dataflow pointcuts. These are propagated statically between expressions to keep track of data dependencies. If an expression matches the pointcut of a dataflow pointcut, it is tagged. This tag is then propagated to expressions which depended on the original expression. The goal is to do as much of the tagging as possible statically, with dynamic methods being used for the remaining tags. This combination minimizes the necessary runtime overhead.

Monitoring properties like the one for SMS spamming (Listing 3), which requires the tracking of abstract state, cannot be supported by DFlow pointcuts, as such properties cannot be expressed as pure information flow.

6 PQL

Martin et al. have proposed a language called PQL (Program Query Language) in which queries about programs can be expressed declaratively [MLL05]. A PQL query matches sequences of method calls and field accesses in a target program. For every match, some user-defined Java code can be executed (optionally replacing the original method call in the target program). The following example calls a privacy checker (line 11) whenever private information (location, IMEI, etc.) is sent out in an SMS message:

```
1  query main ()
2  uses
3    object * privObj, tainted;
4  matches {
5    privObj = TelephonyManager.getDeviceID()
6      | LocationManager.getLatitude()
7      | ... ;
8    tainted := propagateStar(privObj);
9  }
10 replaces sendTextMessage(tainted)
11   with checkAndSend(tainted);
12
13 query propagateStar(object * tainted)
```

```
14 returns object * y;
15 uses
16   object * temp;
17 matches {
18   y = tainted | { temp := propagate(tainted); y := propagateStar(temp); }
19 }
20
21 query propagate(object * tainted)
22 returns object * y;
23 matches {
24   y = tainted.toString() | y = String.concat(tainted) | ... ;
25 }
```

Listing 9: Simple taint propagation in PQL

The `main()` query contains two matching-statements that must both apply for the overall query to match. Firstly, an object (we do not expect a concrete type here) must be obtained by a call to one of the listed functions, e.g. `TelephonyManager.getDeviceID()` for obtaining the IMEI. This object is bound to `privObj` and then followed through various propagation operators, such as direct assignment or string concatenation, using the patterns specified in the `propagateStar` query. If a match is found which is then used as a parameter for a call to `sendTextMessage`, this call is intercepted and the `checkAndSend` function is called instead. This function could then, for instance, ask the user for permission before actually sending the data.

This query shows that PQL supports modularity quite well. `propagate` is a second query that is used to filter executions based on what happens with the value previously matched for the `privObj` variable. The overall `main` query only matches if there is a suitable value for `privObj` returned by a call to one of the listed functions which then also matches the `propagate` query during further program execution. The `propagate` query can be reused in all queries that need to track information propagation.

The example above also shows that PQL supports recursive queries. String propagation may not occur at all (i.e., the variable of the ”get” call is used directly) or an arbitrary number of times. This is implemented using recursive references to the `propagateStar` query in which each reference corresponds to one `propagate` action. Additionally, one should note that matching in PQL works on black-box method calls and field accesses. Therefore, it does not matter whether the body of the called method is implemented in native code or in Java, or whether or not the source code is available, as long as the call site is located in analyzable Java code. In this respect, PQL is superior to many approaches.

7 Lessons learned / tradeoffs

We next briefly outline how well the four languages we studied address the requirements that we identified for the case of security monitoring of Android applications.

Reuse Among all surveyed languages, PQL is the only one that allows reuse of queries and have them call each other recursively. Query reuse is beneficial because it enables sharing of joint sub-queries. This decreases development time and also allows for a more

efficient runtime evaluation, as shared sub-queries need to be evaluated just once. Other languages that support some form of query reuse include PSLang [Erl03].

Recursion Recursion increases expressiveness: Tracematches cannot effectively model information-flow properties because they can only reason about a fixed number of objects. JavaMOP generally shares this limitation; the only escape route is through resorting to plain AspectJ language features. Recursive queries in PQL avoid this problem through a renaming from actual to formal parameters, which effectively allows a query to reason about arbitrarily many objects.

Variable and member access Assume a policy that forbids the sending of any information about calendar entries marked as private. In such a case, the query language must provide a mechanism to restrict tracking to such entries `e` for which a predicate `e.isPrivate()` returns `true`. Tracematches, JavaMOP and Dataflow Pointcuts support such member accesses through a pointcut `if(e.isPrivate())`, PQL does not have such a mechanism. The only way to implement such a policy in PQL is thus to eagerly track *all* calendar entries accessed by the app. If a match is found, the handler can then check `e.isPrivate()` to see if `e` needs to be handled or not. This approach wastes runtime performance, as PQL must track objects which will never need to be handled later on. Furthermore, the filtering step is not part of the language, but of external code, making it harder to reason about the property actually being enforced. It also hinders the implementation of generic handlers (e.g. "ask the user for permission and then continue execution if granted") as the handler code must know the specific objects to filter them appropriately.

Customized monitor state Only JavaMOP allows queries to use custom data structures for tracking internal state. In Tracematches, Dataflow Pointcuts and PQL, matching is fully declarative. Customized monitor state increases expressiveness, which has both advantages and drawbacks. For instance, while the taint-tracking Tracematch from Listing 5 is very fragile with regard to code changes, the JavaMOP specification in Listing 7 is less so: using the custom data structure `taintedStrings`, the JavaMOP specification can track tainted objects recursively, and irrespective of any temporal order. Because any Tracematch specification can only track a fixed number of objects, this is a feature that Tracematch cannot support. The drawback is that custom data structures decrease readability and take away potential for static optimizations. After all, such data structures are outside the control of the specification-language compiler, which therefore cannot possibly reason about them. In the example, it may be beneficial to never add a certain string `s` to `taintedStrings` in the first place if it can be statically determined that `s` will never leak. An ideal language would provide fixed data structures that are expressive enough to support all common use cases.

Global, persistent state Unlike regular desktop programs, Android apps have to adhere to a specific life cycle in which they can be preempted by the virtual machine if the resources taken up by the app are required elsewhere. This life cycle requires monitors to

serialize their state to disk, and resume monitoring when the app's execution is resumed. Another reason for allowing persistent state are properties that span multiple executions. For instance, a policy stating that an app may not send more than 10 SMS a month, no matter how often it was restarted, cannot be enforced without persisting state. None of the surveyed systems has automated support for such persistent state. It could, however, be supported through variable declarations whose values the runtime persists automatically. There are languages (e.g. ConSpec [AN08]) which support such a feature.

Implicit information flow None of the four languages are able to handle implicit information flow. In Listing 2, an attacker can fully reconstruct the IMEI, even though its value does not explicitly flow into any variable sent over the network. There is no direct path from a confidential source to an untrusted sink when considering just data flow. Furthermore, there is no trivial pattern (i.e., the IMEI may not flow into some *toBitStream* function) as the app is considered malicious and may thus employ any kind of code obfuscation. In the general case, the app's source code is not even known to the author of the security monitor. The one approach that would have potential to handle implicit flow is Dataflow Pointcuts. A pointcut `dflow[o,i]` states that data can flow from `i` to `o` without stating *how* data can flow. One could envision an implementation of Dataflow Pointcuts that included implicit flows in the pointcut matching process. For the concrete formalization by Alhadidi et al. [ABB+09], this is however not the case: their formal language does not even contain branching statements syntactically. For Tracematches, JavaMOP and PQL, implicit flows are impossible to handle simply because all approaches match against traces of *explicitly* mentioned events. A possible solution to this problem would be to make implicit flows explicit by exposing them as an event-like primitive `implicitFlow(o,i)`. This primitive would then be implemented similarly to Dataflow Pointcuts. Such a feature would in any case require static analysis beyond pure optimization and could not be handled solely at runtime since implicit flows need to take all possible executions into account and not just one concrete instance.

Primitive data types A similar problem regards the tracking of values of primitive types. All four studied languages can easily track objects, also through assignments between local variables, fields, arrays or even through reflection. This is possible because objects come with an identity, encoded in their object header. But primitive data types pose problems: Assume a user's phone book contains a phone number 12345, and an app sends the number 12345 to an untrusted server. But does this mean that the app is sending the phone number, or is it just coincidentally sending the same sequence of digits? In our running example, an attacker could exploit this ambiguity by obfuscating the code as follows:

```
1 String myObfuscatedCopy(String in) {
2   String resString = "";
3   for (int i = 0; i < in.length(); i++)
4     resString += in.charAt(i);
5   return resString;
6 }
```

Here, `myObfuscatedCopy("12345")` will be equal to `"12345"` but not the same. Moreover, the name of the function `myObfuscatedCopy` is generally unknown. This problem can be solved by two different means. One possible solution is to regard all primitive types as objects, as, for instance, the case in Smalltalk [GR83]. Such an approach, however, could lead to significant performance degradation. A second possible approach is to use totally declarative information-flow specifications such as in Dataflow Pointcuts. In this case, the compiler would generate special code to track a primitive value's identity.

Native code Native code poses a quite similar problem. The contents of a method such as `native String myObfuscatedCopy(String in);` are not ready to be intercepted and analyzed by the security monitor. All four languages allow for manual specifications of the semantics of native calls, by instrumenting code at every call to those native methods. Manual specifications are only viable, however, if the native methods are known, e.g. because they are part of the standard library. Native code under control of the adversary cannot be specified. For such methods, a security monitor should make implicit worst-case assumptions ("The argument might flow into the return value."). None of the four studied approaches implements such a semantics.

Placement of sanitizers A sanitizer is a user-supplied function that converts a sensitive value into an innocent one, for instance by anonymizing, truncating or escaping data. Sanitized values do not require further tracking. It would therefore be beneficial if a query language would allow users to track all data "that has not been sanitized". In PQL, such behaviour could theoretically be emulated using negative patterns. The query should only apply if no sanitize method has been called:

```
1  query main ()
2  uses
3    object * privObj, tainted;
4  matches {
5    privObj = ... ;
6    tainted := propagate(source);
7    ~sanitize(tainted);
8    conn.httpSend(tainted);
9  }
10 executes
11   with Privacy.logViolation(source, tainted);
```

This concept however has several problems. Firstly, PQL can only apply negation to direct method calls or field accesses, not to queries. In the given example, `sanitize` must therefore be the name of a concrete method, no abstraction using query references is possible. Furthermore, the semantics of negated expressions can be quite surprising. It is important to place correct bounds on the scope of the negation. In our example, the query matches when the sink is called without the string having passed an anonymizer after its last propagation. In more complex flows, there might however be multiple positions where sanitization can occur. In such a case, the user would have to explicitly denote all of them in his query to avoid erroneous matches. The other three approaches show similar problems. Possible solutions include (1) fully automated sanitizer placement [LC13], (2) a scoping mechanism, allowing flexible negated queries with an intuitive semantics, and (3)

a mechanism that would allow query programmers to restrict `propagate`-like queries within a set of user-defined sanitizer methods, i.e., define that the `propagate` query never matches inside a set of well-known sanitizer methods.

Support for proceed calls Tracematches and Dataflow Pointcuts support AspectJ-like `proceed` calls. This feature allows the specification to call the originally intercepted event with a possibly updated set of arguments. For instance, the Tracematch in Listing 5 proceeds with sending an SMS message, but first sanitizes the call argument, e.g. by removing or anonymizing private data. (More elaborate sanitizers are possible but outside the scope of this paper.) JavaMOP has no direct support for `proceed`, and neither has PQL. The latter requires users to explicitly enumerate the method calls that are intercepted and the handlers by which they are replaced (cf. Listing 9, lines 10–11). This is more verbose than just using `proceed` and hinders reuse.

Typing of the specification language A language to model (and enforce) security features needs convenient means of specifying data sources and sinks. We must be able to specify the default behaviour of operations that are not explicitly matched in a property. In an adversarial setting, this default would indicate that operations *propagate* tainted information, while if we would like to avoid too many false positives, we would assume information flow by default to *sanitize* data. Through a clear semantics, the language should communicate the intended behaviour clearly. Although a property could be encoded by just matching events and referring to global state (as in aspect-oriented programming in general or in the JavaMOP example in Listing 7), we consider an encoding on the *sequence of events* more readable than a solution where permitted/forbidden sequences of events are implicit.

The development environment for the specification language should adhere to the general concept of *static typing*. On the one hand, this will avoid load- or runtime issues where, for instance, operations in event handlers are badly defined (missing/wrong signatures). On the other hand, we do not seem to gain much flexibility through lax typing. PQL for example allows regular expression-based matches on method names *across classes* in a query, leading to a result of type `Object`. The handler method hence can only extract further information after inspection with `instanceof`.

For improved readability of specifications, (static) types provide a convenient means to *classify* abstract categories of data-types (e.g. “personal data”, “security relevant”), and how operations on them propagate information. Any aspect-based approach may easily take advantage of this. Consider, for example, that the Tracematch specification in Listing 5 needs to track string operations for both the IMEI and the latitude in lines 2 & 3. Both could be subsumed in a case for arguments of type “security relevant” (again ignoring complications through primitive data types). Such a classification is implicitly carried out by means of state in the AspectJ/JavaMOP example in Listing 7.

Note that events do not only correspond to *functions* where the result has a particular property (derived from the parameters). In an imperative language, in addition to the return value, the classification of the callee may also change as a side-effect. For instance,

a class could offer a `sanitize()` method. Whenever this method is called on an object, this object should no longer be tracked.

Discussion As explained above, all of the surveyed languages provide the one or other interesting feature that can be useful for security enforcement. However, no language is ideal, each one is lacking crucial features that are required to make enforcement truly reliable, in particular in light of obfuscated app code. Some of the points discussed above require tool support and will require changes and additions mostly to the backend of the language implementations (native code, primitive types, implicit flow), while others do require substantial modifications to the languages' syntax and semantics.

While we studied a range of languages, there are other specification languages which we are unable to include here for reasons of brevity. Of particular relevance is ConSpec [AN08]. For the purpose of this study, suffice it to say that ConSpec is syntactically and semantically very close to Tracematches and JavaMOP, and presents roughly the same tradeoffs.

For practical usability, it must be sufficiently easy to express common security policies. Therefore, we argue for integrating the features discussed above into a single consistent language that allows flexibility where required, but also saves the user from details where unnecessary. The reuse of subqueries in PQL is a good example of providing such levels of abstraction. Furthermore, the languages need to be implemented efficiently and correctly. During our research, for instance, we found several issues with the PQL implementation, while there is no implementation for the DFlow pointcuts at all.

Ideally, users would be able to deploy their set of specifications onto their applications through the same well-known techniques that are already in use in existing monitoring frameworks.

8 Conclusions & Future Work

In this work we have investigated the suitability of using Tracematches, JavaMOP, DFlow Pointcuts and PQL for enforcing privacy-related security properties on Android apps. As we found, while each language has some interesting and useful elements, none of those languages fully address all requirements for this application area. An ideal language would define a minimal set of language constructs to solve the use cases we discussed. A flexible specification language for information flow properties allows reasoning about security guarantees and, together with instrumentation, the enforcement of such properties.

In the light of our observations, we would like to investigate whether we can also give recommendations with regards to API design. Usage of constructs that complicate a dynamic analysis could be discouraged by "marketplaces" like Google Play and the Apple App Store, which already run analyses on the code for quality assurance and to detect suspicious apps. For example, the `ContactsContract` content provider exposes the entire contents of the contacts. Here, for many applications being able to look up a partic-

ular entry e.g. by name or phone number might already be sufficient, without any need to explicitly crawl the address book.

For the examples in this study, we have mostly considered a "black and white" classification of data, i.e., no sensitive information may leak at all. When augmenting the analysis with user-defined data types, we might as well consider the case of *gradual* or *fractional* measures for quantifying information leakage and enforcing an upper bound on the amount of privacy-sensitive data leaving the user's device. In this case, a user could define an individual balance between privacy needs and application requirements in cases where information leakage cannot be fully prevented without losing functionality.

Acknowledgements This work was supported by the Deutsche Forschungsgemeinschaft within the project RUNSECURE, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the bilateral DAAD/Research Council of Norway project "Runtime Verification for ABS Product Lines".

References

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In [OOP05].

[ABB+09] Dima Alhadidi, Amine Boukhtouta, Nadia Belblidia, Mourad Debbabi, and Prabir Bhattacharya. The dataflow pointcut: a formal and practical framework. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 15–26, New York, NY, USA, 2009. ACM.

[AN08] I. Aktug and K. Naliuka. ConSpec–a formal language for policy specification. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, volume 197 of *ENTCS*, pages 45–58, 2008.

[And12] Android. Android Security Overview, December 2012. `http://source.android.com/tech/security/`.

[BGH+12] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. AppGuard — real-time policy enforcement for third-party applications. Technical Report A/02/2012, Universitäts- und Landesbibliothek, 2012.

[Bit12] Bit9. Pausing Google Play: More Than 100,000 Android Apps May Pose Security Risks, November 2012. `http://www.bit9.com/pausing-google-play/`.

[Cor12] International Data Corporation. Worldwide Quarterly Mobile Phone Tracker 3Q12, November 2012. `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37`.

[CR07] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 569–588, New York, NY, USA, 2007. ACM.

[EGgC+10] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *OSDI*, pages 393–407. USENIX Association, 2010.

[EOMC11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[Erl03] Ulfar Erlingsson. The Inlined Reference Monitor Approach to Security Policy Enforcement. Technical Report 1916, Cornell University, 2003.

[FHM+12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: an analysis of android SSL (in)security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[GCEC12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307. Springer, 2012.

[Goo12] Google Inc. Permissions, December 2012. `http://developer.android.com/guide/topics/security/permissions.html`.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[KB12] Eran Kalige and Darrell Burkey. A Case Study of Eurograbber: How 36 Million Euros was Stolen via Malware, 2012.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–354. Springer, 2001.

[LC13] Benjamin Livshits and Stephen Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, January 2013. To appear.

[MK03] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In Atsushi Ohori, editor, *Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In [OOP05].

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[OOP05] *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[ZZJF11] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on Android). In *Proc. of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107. Springer, 2011.

Using JOANA for Information Flow Control in Java Programs — A Practical Guide

Jürgen Graf, Martin Hecker, Martin Mohr

Programming Paradigms Group
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe
{graf,martin.hecker,martin.mohr}@kit.edu

Abstract: We present the *JOANA* (Java Object-sensitive ANAlysis) framework for information flow control (IFC) of Java programs. JOANA can analyze a given Java program and guarantee the absence of security leaks, e.g. that a online banking application does not send sensitive information to third parties. It applies a wide range of program analysis techniques such as dependence graph computation, slicing and chopping of sequential as well as concurrent programs. We introduce the Java Web Start application *IFC Console* and show how it can be used to apply JOANA to arbitrary programs in order to specify and verify security properties.

1 Introduction

Conventional access control mechanism control what data a program may access, but what happens with this data inside the program, once access has been granted? Information flow control (IFC) aims to answer this question. For example, an email application shall both read data from an address book and send other data to the network, but it should not send address book data over the network. With IFC one can check if the email program may conduct such forbidden behaviour or not.

Much work in the area of IFC has focused either on building theoretical foundations for proveable security guarantees or on practical tools that can detect a subset of all information leaks in a program. We present a static IFC analysis framework named JOANA that aims to combine both directions. JOANA is the first tool that can verify the absence of possibilistic and even probabilistic leaks for full Java bytecode, including exceptions, dynamic dispatch and inheritance. It can deal with sequential [HS09] as well as multi-threaded [GS12, GHMN13] programs and applies to medium sized programs with around 30-50kLoC and in some cases up to 100kLoC [Gra09, Gra10]. A machine-checked proof [WL10, WLS09] guarantees that the underlying algorithms are sound and no potential information flow is missed.

The frontend of JOANA builds upon the WALA program analysis framework[1]. WALA

[1] `http://wala.sf.net/`

comes with an intermediate representation (IR) in SSA-form, a variety of dataflow solvers and a points-to analysis framework. WALA helps to resolve dynamic dispatch, detect potential exceptions and compute side-effects of method invocations. It can deal with Java bytecode, Java source code and javascript programs. Currently we work on support for Dalvik bytecode of the Android platform. JOANA mostly operates on the IR and therefore may be extended to support other languages with relatively little effort.

Our backend is based on *dependence graphs* which capture dependencies between program statements in form of a graph. Those graphs are called *program dependence graphs* (PDG) or, to be more precise, *system dependence graphs* (SDG). Previous work already showed that PDG-based IFC [Ham10] can be useful in practice and has the great advantage that only minimal user effort is needed. This work focuses even more on practicability. We introduce a UI for our IFC framework named *IFC Console* and explain how it can be used to analyze information flow. The source code of JOANA, including IFC Console, is available at `http://joana.ipd.kit.edu` and may be used freely for research purposes.

The major contributions of this paper are:

- We introduce the Web Start application IFC Console that enables developers to check information flow properties of their own programs with little effort.
- We discuss the benefits of using system dependence graphs for IFC analysis in terms of precision and ease of use.
- We discuss the relevant properties for IFC in a concurrent setup.
- Two case studies show how IFC Console can be applied to a single- and a multi-threaded program.

We start with a more detailed introduction to information flow control and show how it can be achieved with the help of dependence graphs in section 2. Then we introduce IFC Console in section 3 and show how it can be applied to sequential and concurrent programs in section 4. Section 5 discusses related work and section 6 concludes the presented work and provides an outlook for future work.

2 Information flow control

Information flow control is concerned with the flow of information inside a program. It is used to prevent leakage of secret information to public output channels, thus to ensure *confidentiality* and it is also used to verify the *integrity* of a program, which is the dual property to confidentiality: It ensures that no unverified input may influence critical computation or secret values. In order to verify these properties, it does not suffice to check where secret or public data is copied from or moved to. In addition, the effects that the value of the data may have on the execution of the program need to be tracked.

For example in figure 1 we do not want an attacker to gain any information about the secret input value by observing the program output. This program contains three `print` statements that produce output. While the statement in line 7 does not reveal any information about

the input, the other two statements do. Line 3 directly prints the value of the input and is therefore called a *direct leak*. The effect of the output in line 5 is more subtle, as it does not print anything related to the input value. However it is only executed if the input is an even number. Hence, the attacker is able to infer that the input is even if he sees the output produced by line 5. These kind of information leaks are called *indirect leaks*.

```
1 void main():
2   int secret = input();
3   print(secret);                    // direct leak
4   if (secret % 2 == 0) {
5     print("secret␣is␣even");        // indirect leak
6   }
7   print("Hello␣World.");            // no leak
```

Figure 1: A program fragment with a direct and an indirect information leak.

An IFC analysis has to detect direct as well as indirect information flow and it needs to know which information is considered secret and what is considered a public output in order to check for confidentiality. In JOANA this is achieved by annotating variables or statements with a *security label*. For the example above we need two different labels: *high* (secret) and *low* (public). Statement 2 is labeled as high input and statements 3, 5 and 7 are labeled as low output. The IFC analysis then checks if any information flow from high input to low output is possible. In contrast to other IFC analyses that are often based on type systems, only statements corresponding to input or output need to be labeled. JOANA propagates the labels for other statements automatically.

This approach is not restricted to only two security labels. For more complex IFC analyses, it supports an arbitrary number of labels. They have to be specified in form of a *security lattice* that defines a partial order on the labels. Any flow from a statement labeled l_1 to a statement labeled l_2 is considered legal iff $l_1 \leq l_2$. In the remainder of this work we will use the standard two-valued lattice $low \leq high$.

2.1 Sequential IFC with dependency graphs

Our IFC analysis[HS09] uses SDGs to conservatively approximate all possible information flow inside a program. A SDG is a language-independent representation of dependencies between statements of a program. It contains nodes for each statement of the program and edges between them if one statement depends on the other one. In sequential programs these dependencies are either direct or indirect dependencies. Direct dependencies, also called data dependencies, occur whenever a statement produces a value, e.g. writes a variable, that the other statement potentially may read. Indirect dependencies between statements occur if the outcome of the execution of one statement decides if the other is executed, e.g. the condition of an if-clause decides if the statements in the body of the if-clause are executed. A machine-checked proof [WL10, WLS09] shows that the SDG is a conservative approximation of the effects of sequential programs and that our IFC algorithm is sound.

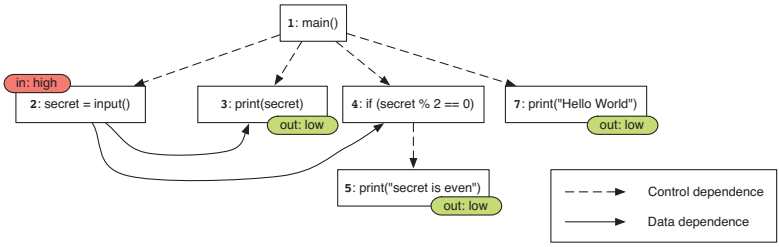


Figure 2: The dependence graph for figure 1 with annotated security labels.

Figure 2 shows a simplified version of the SDG for the program in figure 1. It contains data dependencies between statement 2, where the input is written to variable `secret`, and statements 3 and 4, that read the value of `secret`. Control dependencies between the method entry point and statements 2, 3, 4 and 7 signal that those statements are only executed when `main` is called. Statement 5 however is control dependent on the if-clause in statement 4, because its execution depends on the evaluation of this statement. The input and output statements are annotated with security labels *high* and *low* as described in the previous section.

The SDG based IFC analysis then checks if the graph contains a path from a statement labeled *high* to a statement labeled *low*. To achieve this, we use a special form of conditional reachability analysis that applies *slicing*[Kri03, RHSR94, Wei81] and *chopping*[Gif11, RR95] techniques. This enables us to restrict the set of possible paths in the graph to a subset of *valid paths*, which helps to significantly reduce the number of false alarms. A valid path is a path in the SDG that respects additional conditions, like e.g. context-sensitivity. The example contains two valid paths that correspond to illegal flow: $2 \rightarrow 3$ and $2 \rightarrow 4 \rightarrow 5$. Thus our analysis reports two potential security violations.

In case no violations are found, the program is considered safe. Hence our analysis can guarantee the absence of security violations, but it can only detect the potential presence of leaks, because false alarms are possible due to conservative approximations in our analysis algorithms.

JOANA contains many optimizations that improve analysis precision and thus help to reduce the number of false alarms:

points-to information We use points-to analysis to compute side-effects across method boundaries and to approximate the effects of late binding. Various precision options are available.

exception analysis We include an analysis that detects exceptions that never occur. This is very essential in Java, because almost any instruction may potentially throw an exception, e.g. any object field-access may throw a *NullPointerException* if the referenced object is *null*.

context-sensitive We distinguish between different calls to the same method and offer

unlimited[2] context-sensitivity through interprocedural program slicing [RHSR94].

object-sensitive We distinguish different instances of the same class and methods invoked on different instances.

field-sensitive We distinguish different fields of an object instance through modelling accessible fields in form of an object graph [Gra10].

flow-sensitive The dependencies inside an SDG respect the execution order. It contains only dependencies between two statements $s_1 \rightarrow s_2$ if s_2 may be executed after s_1.

2.2 IFC for concurrent programs

Concurrent Java programs consist of multiple threads that execute in parallel. Threads can communicate through shared variables. In addition to the previously mentioned direct and indirect leaks, these so-called *interferences* between threads introduce two new kinds of information leaks: *possibilistic* and *probabilistic* leaks.

```
1 void thread_1():
2     x = 0;
3     print(x);

4 void thread_2():
5     secret = input();
6     x = secret;
```

Figure 3: Two threads with a shared variable `x` that contain a possibilistic leak.

A possibilistic leak results in illegal flow depending on the order in which statements of different threads are executed. The example in figure 3 has a possibilistic leak. The program consists of two threads that communicate through a shared variable `x`. The `print` statement in line 3 does leak the value of the secret input in line 5 if line 6 is executed after line 2 and before line 3.

```
1 void thread_1():
2     x = 0;
3     print(x);

4 void thread_2():
5     secret = input();
6     while (secret != 0)
7         secret--;
8     x = 1;
```

Figure 4: Two threads with a shared variable `x` that contain a probabilistic leak.

Probabilistic leaks are even trickier. A secret value can potentially influence the probability of the order in which statements that influence public outputs are executed. An attacker that can run the program with the same secret input multiple times is able to infer information about the secret value through a statistical analysis of observable outputs. Figure 4 illustrates this problem. The statement in line 3 prints the value of variable `x`. Depending on the

[2] Multiple recursive calls are not distinguished.

execution order of the statements in line 2 and 8 it prints either `0` or `1`. However the probability that line 8 is executed before the print statement depends on the value of `secret`. The bigger the value of `secret` is, the more time is spent executing the **`while`** loop in lines 6-7 and thus the less likely it is that line 8 is executed before the print statement. So if the attacker observes a huge number of program runs and keeps track of ratio between output `0` and `1` he can infer if the value of `secret` is a large number.

Albeit probabilistic leaks seem to pose more of a theoretical than an actual security thread, this is far from true. With additional knowledge about the scheduling algorithm the attacker is in some cases able to infer concrete values. These leaks have already been successfully used to break well known encryption algorithms [Koc96].

JOANA is able to detect possibilistic as well as probabilistic leaks[Gif12]. It computes possible interferences between threads with the help of points-to and may-happen-in parallel (MHP) analyses. We apply a special version of slicing [Kri03, RHSR94, Wei81] and chopping [Gif11, RR95] algorithms optimized for concurrent IFC. This allows us to achieve precise results that are time-, join- and in future versions even lock-sensitive[GHMN13].

3 IFC Console

IFC Console is a graphical user interface which hides most of JOANA's internals. It simplifies SDG construction and the annotation of SDG nodes with security labels. Instead the user can annotate program artifacts such as attributes, method parameters or bytecode instructions and an integrated heuristic selects the appropriate nodes.

3.1 A Quick tour through the interface

The graphical user interface in figure 5 is divided into two parts. The upper part shows options for SDG construction and general configuration and also contains additional tabs for security label annotation (figure 6) and running the IFC analysis (figure 7). The lower part shows a console that displays detailed output and can be used to enter advanced commands. Every action the user performs is recorded as a command in the console. This allows the user to save his actions to a script file, that can be loaded and automatically replayed.

Configuration Tab The configuration tab in figure 5 is used to select the program to analyze (1.), build or load a SDG for the program (2.), select the security lattice (3.) and save or replay scripts of previous actions (4.).

In order to select a program, the user sets the class path to a directory or .jar file that contains the compiled .class files. Then he hits "update" and selects the `main` method he wishes to analyze in the drop down list.

In the next step the user selects the desired SDG computation options. He can choose how the analysis should handle the effects of exceptions and the treatment of multi-threaded

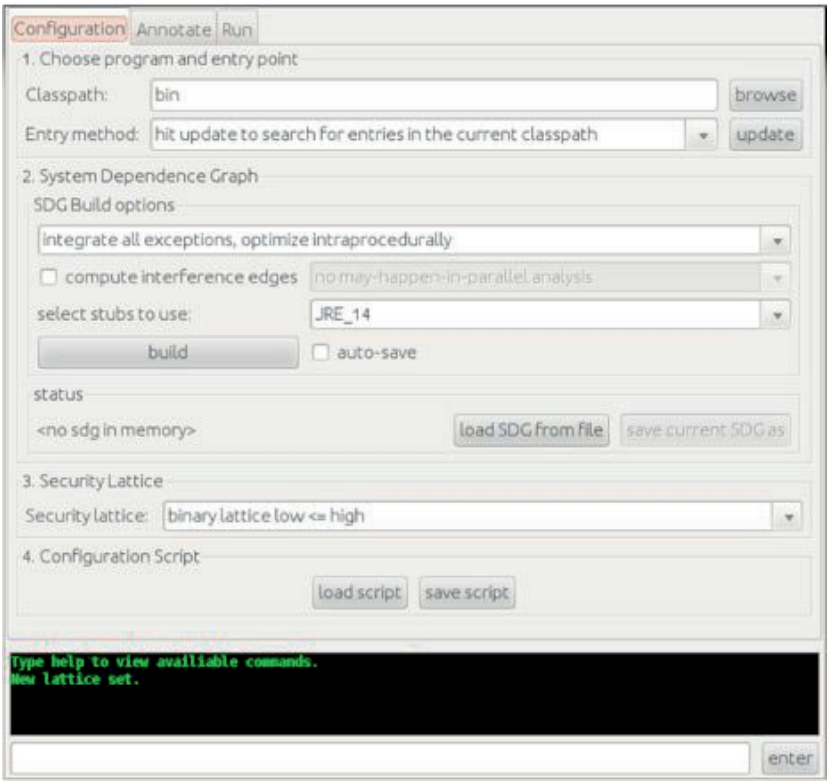


Figure 5: The configuration tab and the console view of the IFC Console.

programs. The exception analysis options are:

integrate exceptions without optimization No additional exception analysis is performed. This is the least precise option that does not detect any impossible exceptions. For example every field access is treated as it may or may not cause a NullPointerException, even subsequent accesses to the same field or references to the `this` pointer.

integrate exceptions, optimize intra-/interprocedurally An exception analysis is performed that detects impossible and also guaranteed exceptions. For example, JOANA identifies field reading accesses where the field can never be null. The interprocedural analysis is more precise but also more time-consuming.

In order to analyze multi-threaded programs, the check box “compute interference edges” has to be selected. Then the user can choose between various precision options of the may-happen-in-parallel (MHP) analysis. The least precise option is no MHP, whereas “precise may-happen-in parallel analysis” takes a closer look at the control-flow of the program and in particular the life span of its threads. For example, it detects that a thread

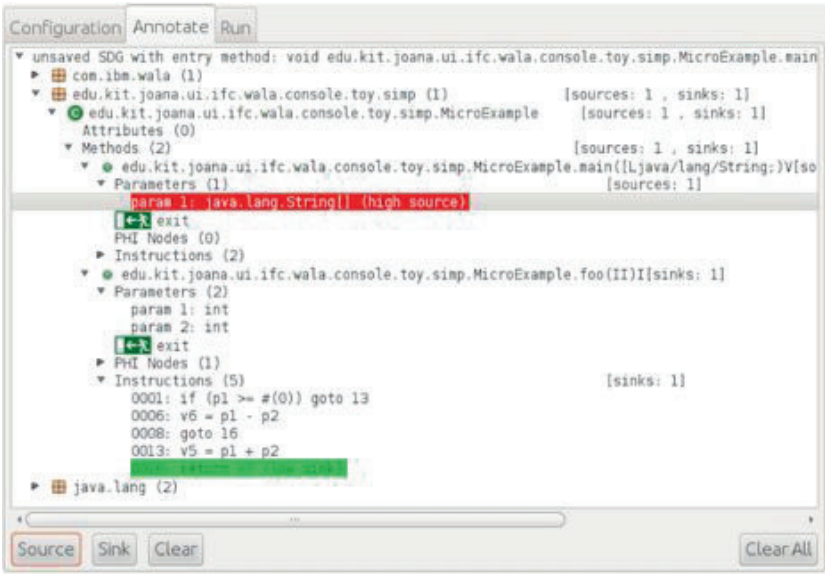


Figure 6: The annotation tab of the IFC Console.

cannot interfere with another thread before it was started or after it has been joined. The user can also choose between *stubs* for different Java runtime environments. Basically, these stubs include predefined models of native methods deep down in the Java standard library. We strongly suggest to analyze JRE 1.4 programs, as later versions of the JRE are far bigger, which leads to an increased runtime of the analysis. When the configuration of the SDG building options is finished, a click on "build" starts the SDG building process.

There is also the possibility to save and load a previously built SDG. The option "auto-save" stores the SDG directly after it has been built to a file with an auto-generated name in the current working directory.

The IFC part of JOANA supports multiple security levels which are arranged in a lattice. The graphical user interface offers three simple lattices, which should suffice for simple cases, the default is the two-valued lattice $low \leq high$.

Annotation Tab The annotation tab in figure 6 provides a tree-like view of the program under analysis. On the top-level the different packages of the program are shown. Unfolding a package shows the classes it contains, unfolding a class shows the attributes and methods it contains and unfolding a method shows its parameters and instructions. Note that not the whole program is shown, but only those parts which are reachable from the selected entry method. A node of the program tree can be turned into an information source or an information sink and annotate it with a selectable security level.

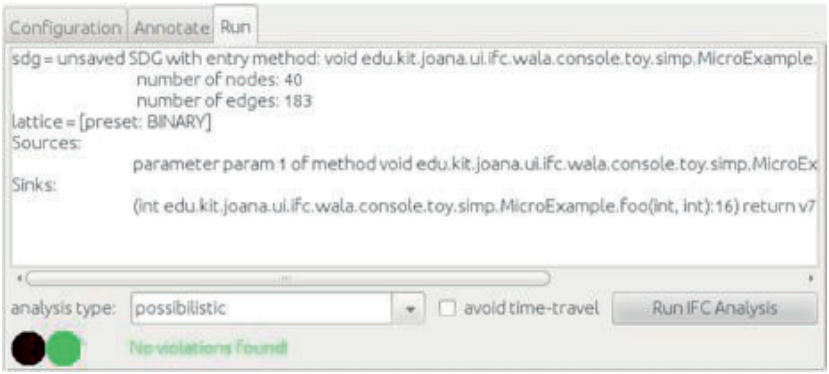


Figure 7: The analysis tab of the IFC Console.

Analysis Tab The analysis tab in figure 7 displays a summary of the analysis options: It shows the size of the SDG, the selected lattice and all annotated sources and sinks. Also it offers to choose between a "possibilistic" and a "probabilistic" IFC algorithm. This is only relevant for multi-threaded programs, for single-threaded programs the default option "possibilistic" is fine. The probabilistic algorithm detects the same leaks as the possibilistic approach and it includes additional probabilistic leaks, which can only occur in multi-threaded programs. Apart from selecting the IFC type, the user can also choose to improve precision by disallowing *time-travels* ([Kri03, Gif12]). This affects both the possibilistic and the probabilistic types. Disallowing time-travels essentially means that the algorithm will discard security leaks made possible only by inconsistent program runs.

The user starts the IFC analysis with the "run" button. When finished, the console part of the GUI shows a report of the detected leaks. Additionally a green light flashes up, if there are no security leaks and the program is guaranteed to be noninterferent. Otherwise a red light signals that potential leaks were detected.

4 Examples

4.1 Sequential IFC - Praktomat

We use a simplified version of the Praktomat system to show how JOANA can be applied to guarantee integrity. Praktomat is a browser based application that allows students to submit their solutions to a given programming task. Then Praktomat runs predefined checks on the submitted solution, e.g. it checks if the code compiles or if the Java Code Conventions are violated. On the one hand this information helps the student to improve his solution and on the other hand these results are also used by the tutor that evaluates the solution later on.

For this example we focus on the way the predefined checks should operate. Their results

are often crucial for the evaluation of the submissions, as manual checks are not feasible for large amounts of submissions and they can also not provide instant feedback to the submitting student. As tutors rely on their results, these checks need to produce fair and reproducible results. A malicious check for example may treat submissions from a specific student in different way then all other submissions - not showing detected failures of this special student.

```
1  public class Praktomat {
2    public static class Submission {
3
4      public int matrNr;
5      public String code;
6
7      public Submission(int matrNr,
8          String code) {
9        this.code= code;
10       this.matrNr = matrNr;
11     }
12   }
13
14   public static class Review {
15
16     public Submission sub;
17     public int failures;
18
19     public Review(Submission sub,
20         int failures) {
21       this.sub = sub;
22       this.failures = failures;
23     }
24   }
26   public static Review runChecks(Submission sub) {
27     int failures = 0;
28
29     if (sub.code.contains("System.err.println")) {
30       failures++;
31     }
32     if (sub.code.contains("catch␣IOException")) {
33       failures += 2;
34     }
35     if (sub.matrNr == 4711) {
36       failures = 0;
37     }
38
39     return new Review(sub, failures);
40   }
41
42   public static void main(String argv[]) {
43     Submission sub = new Submission(2331,
44       "System.out.println(\"Hello␣world.\");");
45     Review r = Praktomat.runChecks(sub);
46     System.out.println(r.failures);
47   }
48 }
```

Figure 8: A simplified version of the automated program submission system Praktomat. It automatically checks submitted programs for predefined failures and helps the tutor to review student submissions. The underlined code violates the security property, as it hides detected failures for a specific student.

We can detect these kind of malicious checkers with the help of JOANA. To achieve this, we specify the information flow property all checkers need to guarantee as follows: The number of detected program failures must not depend on the identity of the submitting student. The code in figure 8 shows a simplified version of the Praktomat system. It contains a class `Submission` to model student submissions and a class `Review` to model the result of the submission checker. The code of the checker is in method `runChecks` in lines 26-40. It is called once from `main` method to perform checks for a single submission. The attribute `matrNr` of class `Submission` stores the identity of the submitting student. We classify this information as *secret* and the `failure` counter in class `Review` as *public*. Then we can use JOANA to verify if the given program is *noninterferent*[GM82] and thus the number of detected failures does not depend on the id of the submitting student. For the given program this is not the case, as lines 35-37 contains a special treatment for the student with the id 4711. JOANA is able to detect this leak. Also when these lines are removed from the program, the checker result no longer depends on the student id and JOANA can verify noninterference for this example.

Using IFC Console We now describe briefly the necessary steps to analyze this example. Specify the appropriate class path, click on the "update" button and select the main method of the class `Praktomat` as entry method. The SDG building options do not have to be changed, so that you can directly build the SDG by clicking on the "build" button. Switch to the annotation tab. Select the attribute `matrNr` of the inner class `Submission` as high source and the attribute `failures` of the inner class `Review` as low sink. In the analysis tab, nothing needs to be configured, since the given program is not multi-threaded, so you can simply run the analysis. As explained before, the analysis finds several leaks. If you remove lines 35-37, you should get no leaks.

4.2 Concurrent IFC - EuroStoxx

Figure 9 shows a program that manages a stock portfolio of Euro Stoxx 50 entries[3]. The program consists of four threads, coordinated by an additional main thread. The program first starts the `Portfolio` and `EuroStoxx50` threads concurrently, where `Portfolio` reads the user's stock portfolio from storage and `EuroStoxx50` retrieves the current stock rates. When these threads have finished, threads `Statistics` and `Output` are run concurrently, where `Statistics` calculates the current profits and `Output` incrementally prepares a statistics output. After these threads have finished, the statistics are displayed, together with a pay-per-click commercial. An ID of that commercial is sent back to the commercials provider to avoid receiving the same commercial twice. The portfolio data, `pfNames` and `pfNums`, is secret, hence the Euro Stoxx request by `EuroStoxx50` and the message sent to the commercials provider should not contain any information about the portfolio. As `Portfolio` and `EuroStoxx50` do not interfere, the Euro Stoxx request does not leak information about the portfolio. The message sent to the commercials provider is not influenced by the values of the portfolio, too, because there is no explicit or implicit flow from the secret portfolio values to the sent message. Furthermore, the two outputs have a fixed relative ordering, as `EuroStoxx50` is joined before `Output` is started. Hence, the program is considered secure.

Using IFC Console Analyzing the concurrent example introduced in 4.2 requires different options because it is multi-threaded. After selecting the appropriate class path and entry method, you have to check "compute interference edges" and choose a MHP analysis. Use the precise MHP analysis, otherwise you will get many false alarms simply because joins are not taken into account.

As mentioned in the example, the portfolio data is secret, so calls to `getPFNames()` and `getPFNums()` in the run method of the class `Mantel00Page10$Portfolio` have to be annotated as high sources. To verify that the secret data cannot influence the commercial messages, which are written into the output referenced by the attribute `Mantel00Page10.nwOutBuf`, it suffices to annotate the calls to its flush methods. These are located in the run method of the class `Mantel00Page10$EuroStoxx50` and in the main method of the class Mantel00Page10,

[3] The description of this program has been taken from [Gif12]

```
1  public class Mantel00Page10 {
2    static class Portfolio extends Thread {
3      int[] esOldPrices;
4      String[] pfNames;
5      int[] pfNums;
6      String pfTabPrint;
7
8      public void run() {
9        pfNames = getPFNames();              // high
10       pfNums = getPFNums();                // high
11       for (int i = 0; i < pfNames.length; i++) {
12         pfTabPrint += pfNames[i]+"|"+pfNums[i];
13       }
14     }
15
16     int locPF(String name) {
17       for (int i = 0; i < pfNames.length; i++) {
18         if (pfNames[i].equals(name)) {return i;}
19       }
20       return -1;
21     }
22   }
23
24   static class EuroStoxx50 extends Thread {
25     String[] esName = new String[50];
26     int[] esPrice = new int[50];
27     String coShort;
28     String coFull;
29     String coOld;
30
31     public void run() {
32       try {
33         nwOutBuf.append("getES50");
34         nwOutBuf.flush();                   // low
35         String nwIn = nwInBuf.readLine();
36         String[] strArr = nwIn.split(":");
37         for (int j = 0; j < 50; j++) {
38           esName[j] = strArr[2 * j];
39           esPrice[j] =
40             Integer.parseInt(strArr[2 * j + 1]);
41         }
42         // commercials
43         coShort = strArr[100];
44         coFull = strArr[101];
45         coOld = strArr[102];
46       } catch (IOException ex) {}
47     }
48   }
49
50   static class Output extends Thread {
51     public void run() {
52       for (int m = 0; m < 50; m++) {
53         while (s.k <= m); // busy-wait sync
54         output[m] = m + "|" + e.esName[m] + "|"
55           + e.esPrice[m] + "|" + s.get(m);
56       }
57     }
58   }
59   static class Statistics extends Thread {
60     int[] st = new int[50];
61     volatile int k = 0;
62
63     public void run() {
64       k = 0;
65       while (k < 50) {
66         int ipf = p.locPF(e.esName[k]);
67         if (ipf > 0) {
68           set(k, (p.esOldPrices[k] - e.esPrice[k])
69             * p.pfNums[ipf]);
70         } else {
71           set(k, 0);
72         }
73         k++;
74       }
75     }
76     synchronized void set(int k, int value) {
77       st[k] = value;
78     }
79     synchronized int get(int k) {
80       return st[k];
81     }
82   }
83
84   static Portfolio p = new Portfolio();
85   static EuroStoxx50 e = new EuroStoxx50();
86   static Statistics s = new Statistics();
87   static Output o = new Output();
88   static String[] output = new String[50];
89   static BufferedWriter nwOutBuf = new BufferedWriter(
90     new OutputStreamWriter(System.out));
91   static BufferedReader nwInBuf = new BufferedReader(
92     new InputStreamReader(System.in));
93
94   public static void main(String[] args)
95       throws Exception {
96     // get portfolio and eurostoxx50
97     p.start(); e.start();
98     p.join();  e.join();
99     // compute statistics and generate output
100    s.start(); o.start();
101    s.join();  o.join();
102    // display output
103    stTabPrint("No.\t␣|␣Name\t␣|␣Price\t␣|␣Profit");
104    for (int n = 0; n < 50; n++) {
105      stTabPrint(output[n]);
106    }
107    // show commercials
108    stTabPrint(e.coShort + "Press␣#␣to␣get␣info");
109    char key = (char) System.in.read();
110    if (key == '#') {
111      System.out.println(e.coFull);
112      nwOutBuf.append("shownComm:" + e.coOld);
113      nwOutBuf.flush();                  // low
114    }
115  }
116 }
```

Figure 9: A possibilistic and probabilisitic secure program from Mantel et al. [MSK07], adapted to Java in [Gif12]. JOANA is the first tool able to automatically proof the absence of probabilistic leaks for this example.

respectively.

In the analysis tab, choose “probabilistic (with precise mhp)” as analysis type. Running the IFC checker yields no violations.

Note, that it is crucial to select “probabilistic (with precise mhp)” as analysis type. If you select “probabilistic (with simple mhp)”, lots of leaks will be found due to many spurious interference edges. Since direct and indirect leaks are included in the probabilistic IFC checker, possibilistic IFC also accepts the program.

5 Related work

Tools for language based IFC and dependence analysis Several other tools for information flow control and dependence analysis of Java programs are available. These tools differ in how much user guidance they require, and which language features they support. Tools like Jif[Mye99, MZZ+01] extend Java with *security types*. In addition to their standard Java type such as `int`, the user annotates variables, fields and method signatures with labels that restrict how information may flow. Jif then checks if these security type annotations are valid and hence if the program is secure. Since Jif supports security type inference only for local variables, in order to check *any* information flow property, the user is usually required to annotate the whole program with security types. It is not enough to only mark those program points where information is read in / written out. For this reason, and since Jif does not support Java features such as concurrency, it is impractical to use Jif or approaches based on Jif[CVM07] with existing code bases.

To alleviate the effort of manually annotating large parts of the program with security type annotations, more elaborate type inference algorithms have been proposed[ST07], but as of yet, there is no practical implementation for full Java.

Similarly to JOANA, the Indus[RH07] tool utilizes several auxiliary analyses to provide SDGs for concurrent Java. These can be used for *slicing*[Wei81], which in turn is used in order to reduce the state space in model checking applications. Unlike JOANA, no explicit support for IFC is provided.

Commercial security scanners like AppScan[4] scan the code of web applications for security vulnerabilities and detect many bugs like, SQL injection, error prone coding practices and other security leaks. Tripp et al.[TPF+09] integrate a taint analysis for javascript into AppScan. Their approach is based on the WALA framework and applies *hybrid thin-slicing*. Their tool scales well and can detect many security leaks in almost arbitrarily large programs with only few false alarms. Due to thin-slicing they miss indirect information leaks and thus cannot guarantee noninterference.

Bodden [Bod12] presents an IFC tool tailored for software production lines that applies the new IFDS/IDE implementation of the SOOT program analysis framework[5]. It can deal with conditionally compiled code in an efficient way and includes a nice GUI. However this tool can only detect a very specific kind of information flow, namely direct leaks through

[4] `http://www.ibm.com/software/awdtools/appscan/developer`
[5] `http://www.sable.mcgill.ca/soot/`

data flow in variables. At this time it cannot detect indirect flow through branches, data flow through heap allocated objects or any kind of concurrency related leaks.

Guarnieri et al. [GPT+11] use a demand-driven taint analysis based on access paths that detects security leaks in javascript websites. They detect direct as well as indirect leaks but lack support for probabilistic leaks. Their approach scales well and can be applied in real world scenarios, but it lacks a sound approximation of the effects of the `eval` function, which is inherently difficult for a static approach.

Therefore Seth et al. [JCSH11] propose a combination of static and dynamic analysis for javascript that can detect illegal information flow in programs with a sound approximation of the `eval` function.

Aside from type-system and SDG based IFC analyses, in [GS05] an abstract interpretation approach to information flow analysis for Java bytecode is proposed. Each bytecode instruction is abstractly interpreted by its direct information flow. Together with the instruction's *scope* (which is similar to control dependencies in SDGs), this is sufficient to obtain a program's information flow. The proposed analysis is not object sensitive and does not handle concurrency.

```
1 if (secret % 2 == 0) {
2     public = 42;
3 } else {
4     public = 42;
5 }
```

(a)

```
1 secret = Math.abs(...);
2 if (secret % 2 == 0) {
3     public = (secret % 2);
4 } else {
5     public = (secret % 2) - 1;
6 }
```

(b)

```
1 if (secret > 17 && secret < 17) {
2     public = 42;
3 }
```

(c)

Figure 10: Semantically secure program fragments

Non-interference and verification of semantic properties Just like JOANA, the tools mentioned so far are imprecise in the sense that they employ a *syntactic* approximation of information flow. Specifically, they will all deem the programs in figure 10 insecure since syntactically the assignments to `public` are dependent on `secret`. Semantically these programs are secure since the value of `public` will not change for different values of `secret`. An attacker who can only observe the value of public variables at the end of the program cannot infer information about the value of secret variables. This base-line notion of security, called noninterference[GM82], applies only to non-interactive, terminating programs, covers no kind of declassification and is overly restrictive for concurrent programs. Hence, a wide variety of security notions have been proposed (cf. e.g. [HS11]).

The analyses described here can be enhanced to infer semantic properties and use these to remove spurious information leak warnings. Such techniques may, however, be computationally expensive and can, in principle, not detect all such properties. The KeY[ABB+05] tool allows the user to manually specify and verify arbitrary semantic properties of sequential Java programs and use them to verify information flow security[BBK+12]. This generally requires a considerable amount of manually provided JML[BCC+05] annotations in the program's source code.

6 Conclusion and future work

We have shown how to conduct information flow analyses on Java bytecode programs using the JOANA IFC Console. To specify an analysis goal, only a minimum of user interaction and no knowledge about the structure of the underlying SDG is required. In the future, we will improve on and streamline the IFC Console usability based on user feedback and experience gathered in the RS3 as well as the KASTEL research program. We are going to offer an API that allows tools such as KeY to employ JOANA as backend for IFC queries that can be answered automatically. In order to deal with large programs, we developed a method for modular SDG computation which helps to analyze isolated components in an unknown context. Within the RS3 priority program, we collaborate with the Software Construction and Verification Group at the WWU Münster in order to improve precision for concurrent programs with `synchronized` methods, e.g. using lock-sensitive interference detection with *dynamic pushdown networks*[GHMN13].

Acknowledgments. This work was funded by the DFG under the project in the priority program RS3 (SPP 1496) and by the BMBF under the KASTEL competence center for applied IT security technology.

References

[ABB+05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, P. H. Schmitt, et al. The KeY Tool. *Software and System Modeling*, 2005.

[BBK+12] B. Beckert, D. Bruns, R. Küsters, C. Scheben, P. H. Schmitt, and T. Truderung. The KeY Approach for the Cryptographic Verification of Java Programs: A Case Study. Technical Report 2012-8, Department of Informatics, Karlsruhe Institute of Technology, 2012.

[BCC+05] L. Burdy, Y. Cheon, D. Cok, et al. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005.

[Bod12] Eric Bodden. Static flow-sensitive & context-sensitive information-flow analysis for software product lines: position paper. PLAS '12, New York, NY, USA, 2012. ACM.

[CVM07] S. Chong, K. Vikram, and A. Myers. SIF: enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[GHMN13] J. Graf, M. Hecker, M. Mohr, and B. Nordhoff. Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks. 1st International Workshop on Interference and Dependence, 2013.

[Gif11] Dennis Giffhorn. Advanced chopping of sequential and concurrent programs. *Software Quality Journal*, 19(2):239–294, 2011.

[Gif12] Dennis Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. PhD thesis, Karlsruher Institut für Technologie, 2012.

[GM82] J. A. Goguen and J. Meseguer. Security Policies and Security Models. *Security and Privacy, IEEE Symposium on*, 1982.

[GPT+11] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. ISSTA '11, New York, NY, USA, 2011. ACM.

[Gra09] J. Graf. Improving and Evaluating the Scalability of Precise System Dependence Graphs for Objectoriented Languages. Technical report, Universität Karlsruhe (TH), 2009.

[Gra10] J. Graf. Speeding up context-, object- and field-sensitive SDG generation. In *9th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010.

[GS05] S. Genaim and F. Spoto. Information flow analysis for java bytecode. VMCAI'05. Springer-Verlag, 2005.

[GS12] D. Giffhorn and G. Snelting. Probabilistic Noninterference Based on Program Dependence Graphs. Technical report, Karlsruhe Institute of Technology, 2012.

[Ham10] C. Hammer. Experiences with PDG-based IFC. In *International Symposium on Engineering Secure Software and Systems (ESSoS'10)*. Springer-Verlag, 2010.

[HS09] C. Hammer and G. Snelting. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *IJIS*, 2009.

[HS11] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Proceedings of the 2011 Marktoberdorf Summer School*. IOS Press, 2011.

[JCSH11] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information flow analysis for javascript. PLASTIC '11, New York, NY, USA, 2011. ACM.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, LNCS. Springer, 1996.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, April 2003.

[MSK07] H. Mantel, H. Sudbrock, and T. Kraußer. Combining different proof techniques for verifying information flow security. LOPSTR'06. Springer-Verlag, 2007.

[Mye99] Andrew C. Myers. JFlow: practical mostly-static information flow control. POPL '99. ACM, 1999.

[MZZ+01] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow, July 2001.

[RH07] V. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer*, 2007.

[RHSR94] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proc. FSE*, SIGSOFT '94, pages 11–20, New York, NY, USA, 1994. ACM.

[RR95] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. *SIGSOFT Softw. Eng. Notes*, 20(4):41–52, October 1995.

[ST07] S. Smith and M. Thober. Improving usability of information flow security in java. PLAS '07. ACM, 2007.

[TPF+09] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, 2009.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81. IEEE Press, 1981.

[WL10] D. Wasserrab and D. Lohner. Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing. In *VERIFY*, 2010.

[WLS09] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-Based Noninterference and its Modular Proof. In *PLAS*. ACM, June 2009.

Type Systems for Domain-specific Languages

Reiner Jung[1] Christian Schneider[2]
Wilhelm Hasselbring[1]
[1] Software Engineering Research Group, Kiel University
[2] Embedded and Realtime Systems Research Group, Kiel University
{rju,chsch,wha}@informatik.uni-kiel.de

Abstract: Model-driven software development employs models to describe different aspects of a system on different levels of abstraction. These aspects are driven by technology or application domain. Modeling is often done in specific graphical or textual notations, called domain-specific languages (DSL). In recent years such languages became very popular in the modeling community to describe structure and sometimes behavior. In the context of type systems, these structures are called types and the behavior is modeled with expressions. The programming language community has developed many concepts to model and specify type systems and the semantics of expressions. However, in the modeling community this is often neglected when specifying meta models and describing their semantics, what may cause problems in developing checks and generators for DSLs. To address this issue, we present an approach, that provides guidance and support during DSL development based on established knowledge on type systems and generator construction, to ease the integration of type systems in DSL. We evaluate this approach with the Xtext language engineering framework.

1 Introduction

Model-driven Software Development (MDSD) uses models to describe different aspects of a system on different levels of abstraction. Some of these aspects are driven by technology, others are desired to obtain mental links to application domains. To compose those models, domain-specific notations are used. These notations can be textual languages, which are also used to serialize and persist models, and graphical notations. Both are largely summarized under the term domain-specific languages (DSL).

DSLs had much an impact on software development in recent years, as models became more important and, mainly, because the tools and frameworks to develop these languages improved significantly. From DSL developers' points of view, a DSL is just a language which allows us to formulate models. The developers focus on the meta model as a set of classes and references, but do seldom apply methods and concepts common in programming language design. However, the developers do employ type systems in their language, even though they are not aware or not caring about that. Often, doing so is even then not considered to be worth the effort when running into serious problems while realizing reference resolution, semantic checks, and transformations.

Xbase [EEK+12] addresses such type system issues, by incorporating the Java type sys-

tem into DSLs and provide partial grammars to use Java types and expressions based on Java types. This limits the DSL to be merely a front end for framework and library APIs, but allows fast development of DSLs including expressions. DSLs, which use other type systems or combine multiple type systems, and DSLs not translated into Java are not supported by Xbase.

Driven by own experience in recent projects [GHH+12, Jun12] and the limitations found in Xbase, we developed an approach to guide development of type systems for DSLs. In this paper, we present this approach and show how it can be implemented with the Xtext DSL framework and tooling [AF11].

Our approach alleviates language maintenance by proposing a partitioning of aspects, artifacts and activities. This partitioning allows to integrate DSL development in agile environments, where feature requests and change of requirements occur very often to suit the usual growing needs.

The remainder of this paper is structured as follows. Section 3 discusses the similarities and differences between meta models and type systems. Section 4 introduces and motivates characteristics of DSL. In Section 5 code generation strategies for DSLs are explained. Section 6 introduces our TS4DSL approach, whose evaluation in two case studies is documented in Section 7. Section 2 discusses related work. And finally, Section 8 summarizes the paper and provides an outlook on our next targets.

2 Related Work

Xbase [EEK+12] is a framework for Xtext [AF11] to leverage the integration of the expressions and the Java type system. It provides, a Java type system meta model and a corresponding expression meta model, following the same distinction in models, as TS4DSL. To ease grammar construction, it provides a comprehensive expression grammar, which can be embedded and customized in Xtext DSLs. However, possible types of an Xbase-based DSL must comply with the Java type system. Thus Xbase addresses developers who use DSLs as front ends for Java APIs and helps them to integrate their DSLs in Java projects. TS4DSL generalizes the Xbase approach and proposes a way on how to develop DSLs with custom or domain-specific type systems.

XTS Völter proposes a powerful framework for realizing type checks. The framework provides a declarative API for contributing necessary information on the DSL's meta model classes and the intended type system semantics. XTS [Vö11] supports primitive types, like **boolean**, **int**, **float**), enumeration types, arrays, and sub-typing relations.

Xsemantics is a DSL to describe the semantics, especially the type system, of Xtext-based languages. The DSL targets developers who want to address typing issues from a formal perspective based on type theory. Xsemantics[1] allows to define judgments and de-

[1] Xsemantics project: `http://xsemantics.sourceforge.net/`

duction rules close to notations common in type systems community (compare [CDJ+97]).

Xsemantics and XTS address both the declaration of type systems for Xtext-based languages [BSVC12]. For simple common typing cases, XTS provides a more compact notation then Xsemantics. However, functionality not handled by the XTS API must be implemented in Java or Xtend, losing the advantage of a declarative specification language. Xsemantics is more flexible and allows to formulate arbitrary type systems. In addition to typing, Xsemantics generates validation checks for Xtext with, so called, check rules.

Both type system approaches can be used with our method of designing a well-typed DSL. Since we focus on the development process and building blocks of a well typed DSL, we do not advocate for a particular type system implementation. In this paper we use Xtend to describe simple typing rules. However, for more complex type systems and better re-use, formal notations should be considered.

3 Meta Models and Type Systems

Meta modeling languages, such as MOF [Obj06] or UML, already discuss typing. However, in many other modeling approaches the type system aspect of meta models is not in the focus of developers.

Meta models are used to describe the structure of software, data and persistence, and in some cases, behavior. They consist of classes with properties, either carrying attributes of base types (also called primitive types) or references to other classes. References are used to express containment and/or to relate to other data objects.

There are a couple of meta modeling frameworks, i.e. tools to create meta models and their implementation in code. The most common is the Eclipse Modeling Framework (EMF) [SBPM09], providing the Ecore meta modeling language, which is an EMOF [Obj06] implementation for Java and the basis for many meta modeling tools. Upon the EMF core many tools have been built, e.g., model-to-model transformation languages like QVT [OMG05], ATL [BDJ+03] and Xtend [AG11], DSL and meta model generators like Xtext [AF11] and EMFtext [HJK+09], storage systems like CDO [Ste12], and many more.

In addition to EMF there are, e.g., the ATLAS Model Management Architecture (AMMA) project group [JBC+06], which is realized on the Eclipse platform, too, and the tools GME [LMB+01] and VMTS [MLC06], which are built on Windows.

Type systems have been used to formalize and reason about the structure and semantics of types. They are used to check the soundness [Pie02, p. 95ff] of type declaration and expressions (also called *terms*).

The basis of all type systems is a set of *base types* [Pie02, p. 117], such as string, integer, float or void. While these types may share portions of their value sets, for the type system they are mutually distinct. In addition to these base types, enumerations and references are also common to many programming languages. Build upon these basic typing elements, *complex types*, such as record, list, array, and class, can be composed [Pie02, p. 129ff].

Besides these structural elements, type systems come with semantic specifications for languages. There exist different formalisms to specify semantics. We selected an operational semantics notation (compare [CDJ+97, Pie02]) as a basis for our semantic specifications. Although they have some weaknesses, e.g., cannot address temporal issues, they are easy to understand by non-practiced developers.

Type systems and meta models, both formalize structure of data and provide mechanisms to describe restrictions on the structure. However, their perspectives are quite different. MOF-based meta models by itself do not provide much restriction mechanisms beside limiting the number of values in an array or limit the type of references. To apply further restrictions, they require a constraint language, such as OCL [OMG06]. OCL is a very powerful functional language, which provides a rich set of functionality, but it is not specifically designed to compute typing constraints.

Type system notations formulate the semantics in form of axioms and rules comprising a set of premises and a conclusion. These rules are closely related to the abstract syntax of a language. They can clearly be separated into rules to ensure type safety for structures and those used to provide the same for expressions. In meta models, it is often hard to separate data structures and behavior, as both are modeled with classes and properties, referencing each other.

4 Domain-specific Languages

Domain-specific languages (DSL) are languages tailored to a specific technical or application domain. Their main goal is to express knowledge (structure and behavior) of a domain in a textual or graphical notation suitable for the domain with less code or at least better comprehensible code than in general purpose languages [MHS05].

DSLs can be distinguished in two main groups. DSLs realized in a host language, which are often implemented as a library providing a coherent set of functions for a domain, are called internal DSLs, while DSLs with an own grammar or other means of notation, are considered external DSLs. DSLs can be embedded in other languages, MPS [VP12] for example focuses on that type of DSL, or are fully self contained languages, like Xtend [AG11].

Beside the specification of structure and behavior, DSLs are largely used to build executable software. This can be realized by interpretation or by transforming DSL code into an executable language code.

In our approach, we focus on human-readable textual, external, self contained DSLs, which are used to generate code for an executable target language. Such textual languages profit from the fact that common version control systems are able to handle differences and merges much better for text than for graphical or other representations. In addition, development environments and generative tools, like Xtext [AF11], provide assistance for automated editor generation and specialized APIs for syntactical and semantic checks, code highlighting and content assists, which allows us to develop the tooling for a DSL quickly and help DSL-users to formulate their specifications.

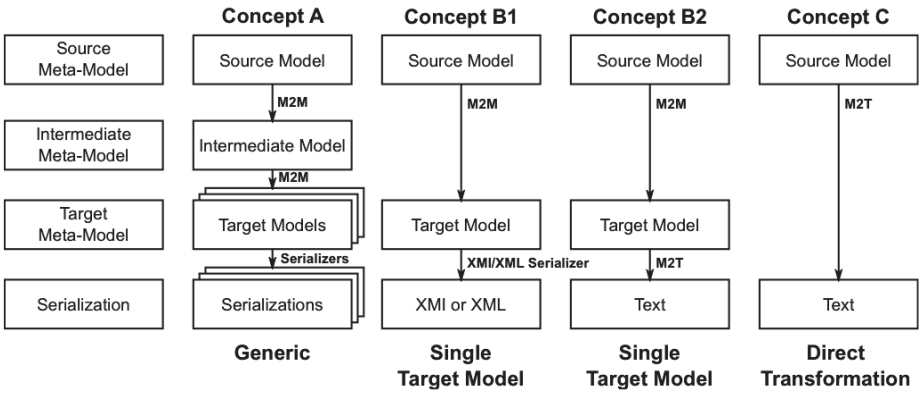


Figure 1: Different generator stack concepts

5 Code Generation Strategies

Generators and compilers scan and parse source code and build an abstract syntax tree (AST) representing the code which is stored in a memory data structure. The AST is then transformed by resolving references (in Xtext this is called linking). The resulting abstract syntax graph may adhere to a meta model for the corresponding language. Or the AST is automatically transformed and stored in the source model. Xtext, the technology we use in this paper, employs an approach where the parse tree is directly transformed into a source model without references resolved, conforming to a AST and in a second step, the references are resolved. Since the concrete steps for obtaining the source model are not in the scope of this work, Figure 1 starts with this source model and omits all previous compiler phases.

The relation between a DSL's meta model (source meta model) and the structure of the generated output can be manifold. Generic code synthesis concepts can be described by four distinct levels (see Figure 1, leftmost column). First, the source meta model, used as storage for artifacts described in a DSL. Second, an intermediate meta model, which resembles a common basis for all target meta models and provides the same type of structural elements to compose types as the target meta models. Third, the target meta models, which are used to represent the generated model in memory. And fourth, the serialization component, which stores target models in files, databases or other means of persistence. Between the three meta models, transformations are used to project the higher level model onto the lower level model.

This expansive structure, well-known in classic compiler construction [ASU88, p. 18], is not always needed. For instance, concepts employing only one target meta model could omit the intermediate meta model. In contexts allowing straight-forward mapping of the source meta model structure to single target code fragments, the target code could be directly generated by a model-to-text transformation. Figure 1 shows four typical DSL-generator model-stacks.

To guide the selection of the different stacks, we established a set of criteria. First, if the target language is a human-readable textual language such as C or Java, not XML or XMI, then the last step can be done by a model-to-text (M2T) transformation. For XML serialization, a meta model for the XML schema definition (XSD) should be generated and used as target meta model. The serialization itself is then done by the XML serializer provided by EMF. For XMI output, the model is serialized with the XMI writer provided by EMF.

Projects using M2T transformations to generate XML encountered serious problems, as M2T transformations write out code in a sequence. While XML supports ids and references to ids (IDREF), references could appear before the id is computed. This requires to implement complex id lookup tables and two pass mechanisms to solve the issue. Using in memory XML models solves the same issue in a more elegant way, as these models could be constructed by model-to-model transformations and then be serialized with XML serializer.

In case the source meta model uses the type system of target meta model, as Xbase [EEK+12] based languages do by employing the Java type system, the source meta model can directly be rendered into a text document.

In case of target meta models, that provide typing structures different from the source meta model, like languages for programmable logic controllers (PLC) [IEC03], or when referencing multiple source meta models, as used in the measure definition language (MDL) of the MAMBA-project [FvHJ+11], more complex composition operations are required. Also, a DSL using sub-typing and dynamic memory allocation with garbage collection, or a target language that supporting just records, then a distinct target meta model is helpful to manage the complexity of the target code generation. This approach is also useful when serializing to XMI or XML, as these serializations make use of cross-references that are hard to keep track of when a source model is directly transformed to text.

6 TS4DSL Approach

TS4DSL is designed to support average DSL developers who use the Xtext framework and tooling to create their languages. These developers often work in an agile environment, provoking changes to meta model and language features in short time. Changes in the meta model and grammar often require changes in typing and validation rules. Such changes to a Java or Xtend-based type and validation system can be complex and time consuming. Furthermore, they cannot be checked for correctness, like formal notations based deduction and reduction rules used to describe operational semantics.

Technologies like, XTS [Vö11] and even more Xsemantics [BSVC12], address the type checking and validation aspects of type system development. While TS4DSL addresses primarily grammar rule design, meta model construction, and generator composition. The approach can be divided in six separate parts. Meta model composition for type systems, integration of base types into the scoping of the DSL, set of Xtext-grammar patterns, type checking, target language integration, and model transformations.

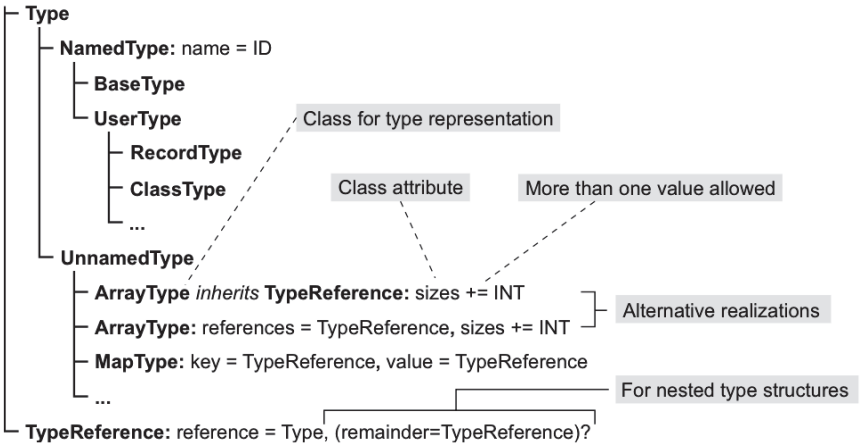


Figure 2: Taxonomy of Types in a Type System

6.1 Meta Modeling

The meta model of a well-typed DSL can be divided into an expression and type part. For most meta models, it is not necessary to place them into different files or packages, but it is helpful to make this distinction while designing the meta model and the grammar. The awareness of these two aspects in language and meta model design helps later to understand language modifications and the transformation development.

In our approach, we use a class Type to model all types. Based on that, a taxonomy of sub-classes for the different type structures can be build. The taxonomy shown in Figure 2, is our general pattern for the composition of type systems. However, in concrete DSLs certain aspects can be solved differently. For example, the distinction of named and unnamed types, can be either modeled by sub-typing or with multiple inheritance.

In order to represent the base types, the class BaseType is used. All types constructible by modelers are subsumed as UserTypes. This distinction will be helpful later on, e.g. while realizing the type resolution and the identifier (cross reference) resolution.

Our approach uses a 'name' field for all NamedTypes including the base types. Alternatively each base type could be represented by a separate type class (compare [Vö11]). The advantage of the former method is, type resolution can handle base types like any other user defined type. In addition grammar rules for type references can use one unique rule to describe all kinds of references to named types.

The type reference, in our approach, is modeled with the class TypeReference. It is employed each time a NamedType is referenced, e.g. in declarations of properties, functions and in the composition of UnnamedTypes. UnnamedTypes are types that cannot be identified by a name, instead they are created in place. In many languages arrays are unnamed types, they are composed of a type reference and size constraints, one for each dimension, or at least the number of dimensions.

6.2 Base Type Integration

Base types are the fundamental elements of a type system. The must be available in a language without definition by the language user. Otherwise, types acting as base types cannot be mapped straight-forward to corresponding typing constructs in the target model. We suggest to represent base types by one dedicated class BaseType in the meta model extending the general NamedType class.

To introduce the concrete base types into the language the base type literals should not be hard-wired in the concrete syntax. While this can be done by exploiting Xtext features, it results in mixing different aspects of the language design in one artifact. Instead, we suggest to define base types in a library that is visible by default. In the EMF and Xtext environment, this is best done by providing a dedicated resource containing the necessary instances of BaseType, one for each concrete base type. This resource should not be persisted as a file or another user accessible resource, as it is an integral part of source model semantics. Therefore, we constructed a virtual resource that is implicitly made accessible to the language tooling[2]. Our approach is similar to the type integration for Java types provided by the Xbase framework. However, it is simpler, as we only have to support base types.

The collection of concrete base types can be determined by means of an enumeration, as we have done in our use cases, or in a String array (see Listing 1). The enumeration approach benefits from the natural mutual disjointness of the literals, while a string array would allow to add the same name twice. Therefore, we recommend an enumerated solution, even though it is a bit more complex than a simple string array.

```
public enum BaseTypes {
    BOOLEAN, INT, FLOAT, STRING; /** base types */

    public String lowerCaseName() {
        return this.name().toLowerCase();
    }
}
```

Listing 1: Base Types of the Use Case *Language for App Development*

The class structure of the base type integration, which has been adopted in part from Xbase' type integration, is shown in Figure 3. The key components of the infrastructure are the previously described BaseTypes enumeration, the TypeResource that is invisibly instantiated while loading a model, the TypeProvider that instantiates the resource and acts as a facade of that resource. The TypeGlobalScopeProvider, employed for determining sets of candidates for resolving cross-references, returns so-called *scopes* that are based on the BaseTypeScope, which contributes the known instances of BaseType. These classes are in charge of making the base types visible in concrete Xtext-based language toolings (specific EMF-based resource factories, textual editors).

[2] An example application can be found in the following git repository: `http://build.se.informatik.uni-kiel.de/de.cau.cs.se.lad.git`

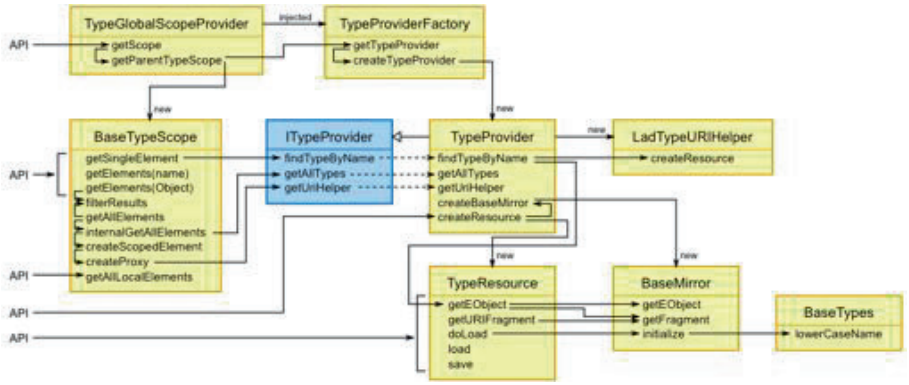


Figure 3: Type System Overview

6.3 Grammar Pattern

Typical rules to construct named and unnamed user types are presented in Listing 2. The first two rules represent the effective taxonomy of the different types. The left most word, is the rule name followed by the corresponding meta model class. If the meta model is automatically generated out of the grammar, the **returns** followed by the meta model class can be omitted. However, for type systems with many different kinds of types, this could lead to meta models hard to comprehend.

```
Type returns type::Type: BaseType | UserType | ArrayType ;
UserType returns type::UserType: ClassType | RecordType ;

TypeReference returns type::TypeReference:
{type::ArrayType} reference = [types::NamedType| ID] ( '[' sizes += INT ']' )+ |
{type::TypeReference} reference = [types::NamedType| ID] ('.' remainder=TypeReference )? ;

MapType: {type::MapType} 'map' '<' key = TypeReference ',' value = TypeReference '>' ;
```

Listing 2: Xtext grammar rules for type representation

After specifying the meta model type, the rule body is defined. The Xtext grammar follows here loosely the EBNF with some extensions. First, names followed by a = are properties of the meta model class associated with the rule. Second, expressions in square brackets are used to express references. For example, in TypeReference the expected token is an ID which is limited to names of type::NamedType of the meta model. Third, meta model classes in curly brackets are used to ensure the creation of an instance of the given type.

```
PropertyDeclaration returns type::PropertyDeclaration:
    modifiers += Modifier type = TypeReference name = ID ;

FunctionDeclaration returns type::FunctionDeclaration:
    modifiers += Modifier type = TypeReference name = ID
    '(' (parameters += ParameterDeclaration (',' parameters += ParameterDeclaration)*)? ')'
    body = Body ;
```

Listing 3: Xtext grammar rules for properties and functions

```
/** axiom case 1: determines the type of 'true' and 'false', similar for IntValue, Enumeration, etc. */
def dispatch Type getActualType(BooleanValue value) {
    return typeProvider.findTypeByName("boolean");
}
/** recursive case 1/axiom case 2: determines the referenced type of a TypeReference by determining the type
  * of the last component of the (potentially compound) declared type, e.g. A.B[5] */
def dispatch Type getActualType(TypeReference ref) {
    return ref.remainder?.actualType?:ref.reference;
}
/** recursive case 2: determines the type of a value, e.g. 'boolean x = 5;', by determining the declared type */
def dispatch Type getActualType(ValueDeclaration decl) {
    return decl?.typeReference?.actualType;
}
/** recursive case 3: determines the type of an identifier by determining the type of the identified element,
  * e.g. of the declared value 'x' */
def dispatch Type getActualType(ValueReference ref) {
    return ref?.reference?.actualType;
}
/** recursive case 4: determines the type of a declared function by determining its referenced return type */
def dispatch Type getActualType(FunctionDeclaration decl) {
    return decl.type.actualType;
}
/** recursive case 5: determines the type of a function result by determining the called function's return type */
def dispatch Type getActualType(FunctionCall call) {
    return call.functionRef.actualType;
}
/** recursive case 6: determines the type of an assignment by determining the modified value's type */
def dispatch Type getActualType(Assignment assignment) {
    return assignment.target.actualType;
}
```

Listing 4: Type resolution realization by means of Xtend's dispatch extensions (excerpt)

Listing 3 shows general patterns for property declarations as well as the definition of functions, methods, procedures or any other parametrized structure. In many languages, those elements can have modifiers such as **public** or **static**. The subsequently required part TypeReference specifies the return type of definition, the **name** attribute its identifier. The non-shown rule ParameterDeclaration is similar to that of PropertyDeclaration, it merely employs different modifiers. Furthermore, properties are often initialized in its declaration. To enable that the rule PropertyDeclaration must be extended by an optional call of the rules Expression or Literal whose result is assigned to a field called expression or value.

6.4 Implementing Type Resolution

Occurrences of types in declarations are represented by a TypeReference maintaining a non-containment reference to the actual type. Applying this delegation pattern is reasonable, as the reference resolution is limited to instances of TypeReference instead of providing one for each of the available kinds of declaration. In our Xtext-based setting this resolution is realized in terms of a *scope provider* that is supported by the former mentioned TypeGlobalScopeProvider. The implementation follows the usual Xtext scope provider declaration scheme [AF11].

In order to establish type checking of expressions of a DSL, like type compatibility of left and right hand side of assignments or operands of binary operations, we employed

the Xtend language. Its features like *null-safe feature call* and *method dispatch* proved very comfortable and allow the compact implementation outlined in Listing 4. Note that ValueDeclaration is a super type of PropertyDeclaration.

Xtend allows a very compact formulation, e.g. **null** checks are expressed by the question marks. Second, Xtend allows suffix notation in case the first parameter of the method, here called *extension*, is type compatible. Using this feature, method name prefixes like 'get' can be skipped, see calls of actualType alias getActualType. Third, the keyword **dispatch** instructs Xtend's code generator to create additional type inference code w.r.t. the extension parameter types. Hence, when calling getActualType(...) for some expression, assignment, or value declaration the extension requiring the most special fitting parameter type is chosen. Thus, by means of such partial inference rules that are dedicated to particular declaration and expression types and that delegate to each other, the type system semantics can be realized very precise, easy to understand and easy to extend. As mentioned in Sec. 2 a dedicated type system framework can by applied alternatively.

6.5 Target Language Integration

The previous sections focus on development of the DSL, the source meta model and the code relevant for editors and source model composition. This section addresses the integration of the target language and target meta model. The simplest method to generate code in a target language is a model-to-text transformation, which is covered by the Xbase framework and well documented and explained in related tutorials [EEK+12].

In this paper, we discuss the more complex target code genesis through transformations into intermediate or target models. The differentiation between an intermediate or target meta model, as shown in Figure 1, is only of a contextual nature, because an intermediate model is subsequently transformed into another model, while the target model, is serialized into text or other kind persistence with a model-to-text or respectively model-to-persistence transformation. These last transformation from a model to a persistence technology is covered by Xtend [AG11] as a model-to-text transformation language and serialization components provided by EMF [SBPM09]. We focus therefore, on the target meta model construction as a requirement for the model-to-model transformation.

The composition of a target meta model requires knowledge of the structure of the type and expression system of the target language, and in addition, knowledge of the storage system, especially when the target model is stored in a database, XML or XMI-file. The retrieval of type and expression system information of a language can be a complex and time consuming task. Therefore, it is good advice to look for existing meta models first. The Eclipse modeling project[3] provides meta models to cover OCL. Java and Xtend meta models can be found in the Xtend repositories[4]. In other scenarios, an XSD might be available to describe the language and its storage model. For example, special computers for automation processes, called programmable logic controllers (PLC), are programmed

[3] http://eclipse.org/modeling/mdt/

[4] http://git.eclipse.org/c/xtend/org.eclipse.xtend.git

often languages specified in IEC EN 61131-3 [IEC03]. A storage meta model for these languages, formulated in an XSD, is provided by the PLCOpen group[5]. Such XSD can automatically be transformed into an EMF meta model[6]. As the PLCOpenXML specification does not handle the textual languages of the standard, this portion has to be added by hand. However, the meta model genesis from the XSD is still a big time saver, as the graph and structural based languages of the IEC EN 61131-3 are covered.

For some languages, there might not be a meta model or schema available. In that case, it has to be constructed from scratch. To start the construction, the base types of a language and the set of compositional typing rules have to be determined. The meta model for a target language type system is then constructed in the same way, as the DSL's meta model, honoring the collected typing rules.

The resulting target meta model, contains a type system taxonomy, conceptually following the structure from Figure 2, tailored for the target language. It is important to restrain the meta-model to the properties of the target language. Any addition makes the meta model more complex and harder to understand. Furthermore, such additions are often triggered by separate concerns and should therefor be modeled in a separate meta model. However, the target meta model type system can be simpler than the target language type system to restrain transformation development.

6.6 Type Transformations

The transformation of a source model into a target model is the central element of every DSL. The transformation and the semantics of the target language determine the implemented semantics of the DSL, which must adhere to the semantics specified in the DSL design. We identified four relevant activities to guide the transformation development. First, the partitioning of the transformations. Second, realization of source language concepts. Third, mapping of base types. And fourth, the mapping of complex types.

The primary separation of transformations is along type structure and expression. If a language has many different type structures, it is advisable to implement them in separate artifacts. Transformations for expressions can be divided along expression kinds, such as literals, unary and binary operators, method and function calls, or statements. With Xtend as transformation language, the partitioning is best realized by separate Xtend-classes.

Source language concepts which are not present in the target language, like garbage collection, synchronization or memory management, must be provided by a runtime system. Therefore these concepts must be identified, an API must be specified and implemented, and finally used in the transformation.

The mapping of base types is achieved by collecting them for source and target meta model together with their ranges and limitations. Even if types in source and target meta model have similar names, they can be quite different. For example, in Java int is a 32-bit signed

[5] http://www.plcopen.org/

[6] http://yoxos.eclipsesource.com/places/node/org.eclipse.xsd.ecore.converter.feature.group

integer, while in IEC EN 61131-3 INT is 16-bit signed integer and the corresponding type for the Java type would be DINT.

The mapping could become more complex, when a source meta model type, like interval, cannot be directly mapped to a compatible structure in the target meta model. For example, an interval type, based on int, has lower and upper bounds defined by arbitrary integer values. Furthermore, it may either cause an runtime exception when one bound is exceeded, or wrap, like integers do in CPU architectures. Such type can be mapped to a target model base type, however to completely realize the semantics in the target model, the identified assertions must be covered.

To ensure boundaries of source model elements in the target model, all expressions, which compute data must be identified, and checked whether they can cause a transition from a valid state to a state prohibited by the source language semantics. For example, an interval increment cannot be transformed just to a C increment operator variable ++, it must be embedded in an function either triggering an overflow error or implementing a value wrap.

In some cases, the type of the target language is more limited than the type of the source language. A typical case are strings in Turbo-Pascal, which use the first byte of a string to store the length of the string. Therefore, strings cannot exceed 255 characters. Source languages realizing strings without or with a greater capacity limit cannot be mapped to a Pascal string. A solution for this particular case are arrays or null terminated buffers, which are only limited by the underlying hardware model. To implement strings on that basis, every functionality of the source language must have a corresponding API operation.

Complex types can often be mapped to a combination of record or variant types. Of special interest are the transformation of sub-typing and sub-classing to type systems not supporting such facilities. One strategy is to use records holding references to parent types, as implemented in Objective-C [App09], or expand all types and generate records for each type. The first, method has its advantages, however, it requires references or pointers for its realization. Especially, method-calls are mapped to function pointers and held in lookup tables. For environments without pointers, like the IEC EN 61131-3, sub-typing can be realized by the second method.

7 Evaluation

We developed TS4DSL out of experiences made in different DSL development projects and the Xbase project. The evaluation and refinement of the approach was primarily executed in the MENGES project [MEG09] evolving the MENGES-DSL [GHH+12]. Subsequently, we used our approach to develop languages in context of MAMBA [FvHJ+11].

MENGES The project goal was to develop a DSL and tooling for electronic railway control centers utilizing industry grade PLCs. PLCs are often programmed in languages specified in IEC EN 61131-3 [IEC03]. The languages share a common type system, which provides a wide range of integers, bit vectors, records and intervals. It does not provide

variant or unit types, dynamic memory management, pointers or references. The latter three are considered dangerous for reliable real-time systems.

The language was developed in an agile environment, where present methods of our project partners were formalized into a specification language. In parallel a code generator was implemented. This lead soon to a hard to maintain generator, grammar and type resolution. To solve these issues, we integrated compiler construction methods into the agile DSL process, which allowed us to continue the development in an agile fashion, by introducing new programming concepts for language users, evaluating these concepts, and modifying them according to user requests.

MAMBA The MAMBA-project addresses software measures and the interpretation of measurements taken during static or dynamic analysis of software. The measures and observations are modeled in an EMF meta model for SMM [Obj11], which is hard to compose by hand. As a solution, two languages the Measure Definition Language (MDL) and the Measure Query Language (MQL), have been developed. An SMM model can reference any EMF model representing an executable specification. Measures in an SMM model reference nodes in an specification expressed by a scope query in OCL. Therefore, MDL and MQL must be able to support the various type systems used by these specifications. The measures declared in MDL return values conforming to types declared in SMM, which consist of a base type and a unit, e.g., count, meter or time. These measures can be used in equations and are instantiable in MQL. They can handle values based on the SMM type system or the type system of the language used to specify a software system.

Early versions of the languages avoided the complexity of the combination of multiple type systems, which made type checks in expressions impossible. Errors could only be detected at runtime. Errors in scope expressions might even be overlooked and remain undetected. Our approach helped to develop better type checks and honor the various involved type systems, which make MDL and MQL applicable for bigger measuring models.

The present application of TS4DSL showed that using type system concepts are helpful to develop DSLs in a safe way and keep them maintainable while neglecting those concepts can lead to languages hard to maintain.

8 Conclusion

TS4DSL allows us to develop more complex DSLs, which exceed simple API front end DSLs. It enables DSL developers to think and use type systems in their DSLs, which allows them to implement checks and generators in a structured way. In our case studies, TS4DSL improved the quality of the developed languages and helped to keep them maintainable and extensible for future developments. Software projects, which use multiple languages to address different aspects of an application, can profit from our approach, which explicitly incorporates type systems in the DSL development process.

The development of type resolution and subsequently, type checking, with Xtend can be

made in a more compact way as in Java itself. However, it is only loosely coupled with the grammar description in Xtext and algorithms have to be repeatedly implemented for every language. Therefore, the incorporation of a suitable notation for semantics and type resolution into our approach, like Xsemantics or XTS is part of our future work. In addition, we want to discuss further type concepts, like let and generics, and provide guidelines and best practice for their realization in an public accessible way. Furthermore, a better integration of syntax and semantics would be preferable, as well as better ways to compose languages and meta-models out for grammar fragments.

References

[AF11] Itemis AG and Eclipse Foundation. XText - DSL devlopment framework. Website `http://www.eclipse.org/Xtext/`, 2011.

[AG11] Itemis AG. Xtend 2. Website `http://www.eclipse.org/Xtext/documentation/2_0_0/01-Xtend_Introduction.php`, 2011.

[App09] Apple Inc. *Objective-C Runtime Programming Guide*, October 2009.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau, Teil 1*. Addison-Wesley, Bonn, 1988. englische Originalausgabe: Compilers—Principles, Techniques and Tools, 1986, 1987 by Bell Telephone Laboratories, Inc. übersetzt von Gerhard Barth und Mitarbeiter.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[BSVC12] Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo. Approaches and Tools for Implementing Type Systems in Xtext. In *Software Language Engineering*, Lecture Notes in Computer Science. Springer, 2012. To Appear.

[CDJ+97] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[EEK+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: implementing domain-specific languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 112–121, New York, NY, USA, 2012. ACM.

[FvHJ+11] Sören Frey, André van Hoorn, Reiner Jung, Wilhelm Hasselbring, and Benjamin Kiel. MAMBA: A Measurement Architecture for Model-Based Analysis. Technical Report TR-1112, Department of Computer Science, University of Kiel, Germany, December 2011.

[GHH+12] Wolfgang Goerigk, Wilhelm Hasselbring, Gregor Hennings, Reiner Jung, Holger Neustock, Heiko Schaefer, Christian Schneider, Elferik Schultz, Thomas Stahl, Reinhard von Hanxleden, Steffen Weik, and Stefan Zeug. Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke. In Stefan Jähnichen, Axel Küpper, and Sahin Albayrak, editors, *Software Engineering*, volume 198 of *LNI*, pages 119–130. GI, 2012.

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2009.

[IEC03] Deutsche Kommission Elektrotechnik Elektronik Informationtechnik im DIN und VDE, Beuth Verlag, Berlin. *IEC EN 61131-3*, 2003-12 edition, 2003.

[JBC+06] Frédéric Jouault, Jean Bézivin, Charles Consel, Ivan Kurtev, and Fabien Latry. Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In *Proceeding of the 1st ECOOP Workshop on Domain-specific Program Development (DSPD)*, July 3rd 2006.

[Jun12] Reiner Jung. Introducing Type-Systems in Xtext-Languages. Website `http://se.informatik.uni-kiel.de/news/introducing-type-system-in-xtext/`, October 2012.

[LMB+01] Ákos Lédeczi, Miklós Maróti, Árpád Bakay, Gábor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Péter Völgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing*, 2001.

[MEG09] Verbundprojekt 5 im Kompetenzverbund Software Systems Engineering. Technical report, CAU, Institut für Informatik, 2009.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[MLC06] Gergely Mezei, Tihamér Levendovszky, and Hassan Charaf. Visual Presentation Solutions for Domain Specific Languages. In *Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, Austria, 2006.

[Obj06] Object Management Group (OMG). Meta Object Facility (MOF) Specification 2.0 Core. formal/2006-01-01, April 2006.

[Obj11] Object Management Group, Inc. Architecture-Driven Modernization (ADM): Structured Metrics Meta-Model (SMM), V. 1.0 Beta 3. `http://www.omg.org/spec/SMM/`, 2011.

[OMG05] OMG. *MOF QVT Final Adopted Specification*. Object Management Group (OMG), June 2005.

[OMG06] OMG. *Object Constraint Language Object Constraint Language, OMG Available Specification, Version 2.0*. Object Management Group (OMG), 2006.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[Ste12] Eike Stepper. Connected Data Objects (CDO). Website `http://www.eclipse.org/cdo/documentation/index.php`, seen November 2012.

[VP12] Markus Voelter and Vaclav Pech. Language modularity with the MPS language workbench. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1449–1450, Piscataway, NJ, USA, 2012. IEEE Press.

[Vö11] Markus Völter. Xtext/TS - a Typesystem Framework for Xtext. Website `http://www.infoq.com/articles/xtext_ts`, 2011.

From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems

Jan Oliver Ringert* and Bernhard Rumpe and Andreas Wortmann

Software Engineering
RWTH Aachen University
Ahornstrasse 55
52074 Aachen, Germany
http://www.se-rwth.de/

Abstract: Software development for Cyber-Physical Systems (CPS) is a sophisticated activity as these systems are inherently complex. The engineering of CPS requires composition and interaction of diverse distributed software modules. Describing both, a system's architecture and behavior in integrated models, yields many advantages to cope with this complexity: the models are platform independent, can be decomposed to be developed independently by experts of the respective fields, are highly reusable and may be subjected to formal analysis.

In this paper, we introduce a code generation framework for the MontiArcAutomaton modeling language. CPS are modeled as Component & Connector architectures with embedded I/O$^{\omega}$ automata. During development, these models can be analyzed using formal methods, graphically edited, and deployed to various platforms. For this, we present four code generators based on the MontiCore code generation framework, that implement the transformation from MontiArcAutomaton models to Mona (formal analysis), EMF Ecore (graphical editing), and Java and Python (deployment). Based on these prototypes, we discuss their commonalities and differences as well as language and application specific challenges focusing on code generator development.

1 Introduction

Cyber-Physical Systems (CPS) [Lee06] are distributed interactive systems which combine computational and physical processes. Typical CPS are found in the manufacturing, automotive, smart energy, avionics, and distributed robotics domains. Software development for CPS is a sophisticated endeavor which yields many challenges. The systems are logically and physically distributed, need to perform on diverse platforms, fulfill certain run-time properties, and deal with communication issues.

Component-based software engineering has been applied to tackle these complexities by breaking systems down to platform dependent components [BBC+07, NFBL10], thus requiring domain experts to be expert software developers, too. The "accidental complexi-

*J.O. Ringert is supported by the DFG GK/1298 AlgoSyn.

ties" [FR07] arising from this gap between problem domain and implementation domain can be reduced using modeling techniques [SSL11, BR12].

We propose a model-driven approach to engineering of CPS where the systems are modeled as Component & Connector (C&C) architectures using automata to describe the components behavior. These models can be refined from specifications to implementations (supported by formal analysis techniques) and translated into various platform specific implementations. Our approach combines concepts from architecture description languages (ADLs) for modeling the structure of software architectures [TMD09] and behavior description languages [Har87, GBWK09].

In this paper, we introduce a code generation framework for the modeling language MontiArcAutomaton [RRW12]. MontiArcAutomaton extends the ADL MontiArc [HRR12] with behavior description by embedded I/O$^{\omega}$ automata [Rum96, Mon12]. This language allows distributed and target platform independent modeling of both the structural architecture of the system as well as its behavior. The languages MontiArc and MontiArcAutomaton are developed using the MontiCore framework [KRV10]. We claim using the MontiArcAutomaton modeling language for the development of CPS yields several advantages:

- The use of a C&C architecture description language makes communication and dependencies explicit in the models.
- MontiArcAutomaton allows behavior underspecification in two forms: (1) incompleteness of triggers to only regulate the reaction to inputs of interest and (2) nondeterministically overlapping triggers to restrict possible behavior as desired. These powerful specification mechanisms are supported by automatic verification and refinement checking as presented in [Kir11].
- The logical decomposition of MontiArcAutomaton components allows independent, incremental and bottom-up modeling, and analysis by different domain experts.

In various robotics projects we have developed MontiArcAutomaton (code) generators for EMF Ecore[1] for graphical editing within Eclipse, Mona [EKM98] theories for verification and refinement checking of models and requirements [RRW12] during the development process, and Java and Python code generation to deploy the modeled systems to robots running the educational LeJOS[2] and the industrial ROS[3] platforms. These generators implement the template-based code generation approach of MontiCore [Sch12], which facilitates development of new target language code generators by allowing to reuse great parts of existing code generators.

We illustrate the benefits of model-based development of CPS with MontiArcAutomaton by describing a system of connected robots providing a collaborative mapping service in Section 2. Afterwards, we introduce the MontiCore framework in Section 3 and we describe the MontiArcAutomaton modeling language in Section 4. Subsequently, we introduce and discuss the different code generators in Section 5. We review related work in Section 6 and conclude this contribution in Section 7.

[1]The Eclipse Modeling Framework Project: `http://www.eclipse.org/modeling/emf/`
[2]Java for LEGO Mindstorms `http://lejos.sourceforge.net/`
[3]Robot Operating System `http://www.ros.org/`

2 Example Software Architecture and Behavior Implementation

An engineer is developing a system of distributed mobile robots to map an a priori unknown office floor. To lower production cost by avoiding expensive sensors, the robots have to estimate their position based on their starting point and performed movement commands. This naive odometric approach to simultaneous localization and mapping (SLAM) [TL08] is prone to produce inaccurate maps, as the difference between estimated position and real position increases over time. The engineer counters this problem by implementing a simultaneous and cooperative discovery by multiple robots communicating via Bluetooth.

The engineer developing the robots has defined a system architecture based on the sensors and actuators available. Figure 1 illustrates the architecture of a single SLAM robot consisting of a front mounted `TouchSensor` component to detect obstacles, a `BumpControl` component implementing the robot controller, a `MapBuilder` component constructing the map from commands passed to the motors and feedback from other SLAM robots received via the `Bluetooth` component.

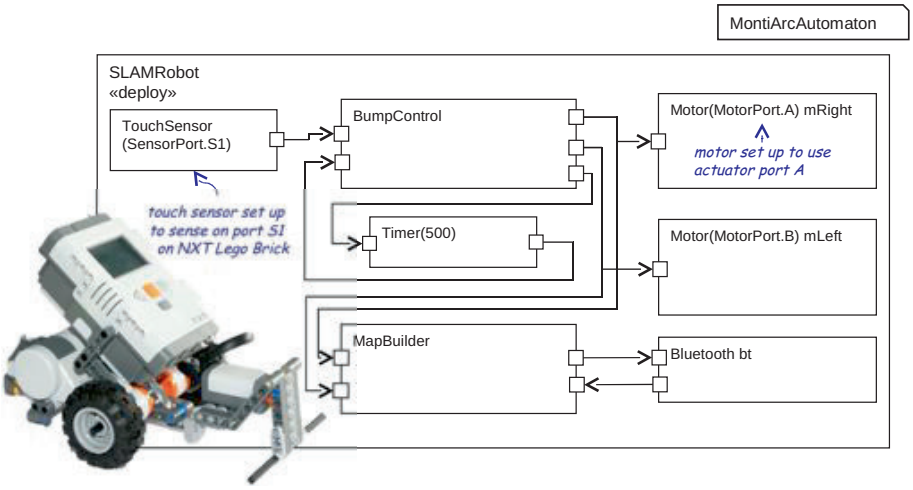


Figure 1: The SLAM robot architecture `SLAMRobot` with a `TouchSensor`, a `Timer`, two motors, the controller `BumpControl`, the component `MapBuilder`, and the Bluetooth communication.

The robot will drive around straight forward until it discovers a new boundary of the map to be explored — by bumping into it. It will then back off and continue the exploration continuously monitoring its own position and communicating with the other robots. For the rest of the example, we focus on the component `BumpControl` that handles the bumping into map boundaries and the subsequent driving maneuvers.

After developing the system architecture, the engineer starts designing the implementation by creating an initial version that defines basic behavior constraints for the `BumpControl` component (e.g., do not drive forward when the bumper is pressed). She later refines it to

the implementation shown in Figure 2. According to the implementation the `BumpControl` component starts in state `idle` with both motors stopped. Once the bumper is pressed by the user to activate the robot, it sends a `FORWARD` command to both motors. When the SLAM robot runs into an object, it backs up, turns around and proceeds forward. The backing and turning times are determined by an external timer that is set via a message on port `tc` and responds with an alert via port `ts`.

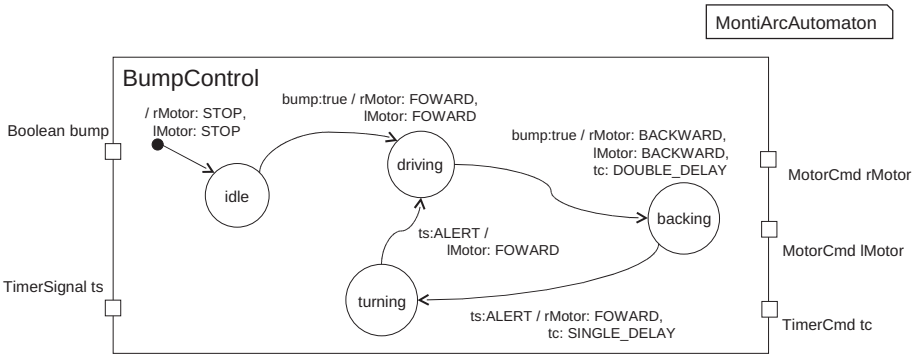


Figure 2: The MontiArcAutomaton component `BumpControl`.

Both the initial specification and the refined implementation are translated to Mona to check refinement, i.e., to check that the implementation does not violate its specification. An excerpt from the translation of the component `BumpControl` is shown in Listing 3.

Mona

```
1  pred bumperbot_BumpControl( var2 bump_true, var2 bump_false,
2    # ..., more port values here
3    var2 allTime )
4    # states of automaton
5  = ex2 idle, driving, backing, turning:
6    # ... constraints: one state at a time
7    # initial states and their outputs:
8    ( (0 in idle & 0 in rMotor_STOP & 0 in lMotor_STOP ) &
9    all1 t: t+1 in allTime => (
10     (t in idle & t in bump_true &
11      t+1 in driving & t+1 in rMotor_FORWARD &
12                       t+1 in lMotor_FORWARD ) |
13     # ... more transitions here
14   ));
```

Listing 3: An excerpt from the Mona predicate generated from the automaton inside `BumpControl`.

To analyze the model a MontiArcAutomaton code generator translates each component to a predicate over streams of messages. The parameters of the predicate are the possible values on input and output streams of the component over time (see Listing 3, l. 1 for

the stream on input port bump) and a synchronized time variable allTime (l. 3). The encoding of message streams into WS1S logic (Weak Second-order monadic logic of 1 Successor) is inspired by [Sch09]. It uses one second order variable for each possible value on a stream, e.g., bump_true and bump_false (l. 1). The predicate holds iff the output streams are valid responses to the input streams.

Values on streams and active states are represented by sets of natural numbers, e.g., sets idle and driving (l. 5). A number $t \in \mathbb{N}$ is in a set iff the corresponding state (or value) is chosen at time t. Thus the initial state idle is translated to `0 in idle` (l. 8). The transition system is defined analogously for the source state and input at time t and the target state and output at time t+1 (ll. 10-12). For more information on our translation to Mona see [Kir11].

After making sure that the implementation refines its specification the engineer is confident about her work and generates Java code that she compiles and deploys to her robot. Parts of the Java code generated from component BumpControl are shown in Listing 4. The displayed excerpt of the Java class BumpControl shows how the automaton's initial state (l. 3) and the initial values of the motors (ll. 4, 5) are set based on the initial output as shown in Figure 2. The excerpt of the compute() method shows how the first transition from state idle to state going is translated to Java.

Java

```
1  public class BumpControl implements Component {
2    public void init() {
3      this.state = State.idle;
4      this.rMotor.setCurrentValue(MotorCmd.STOP);
5      this.lMotor.setCurrentValue(MotorCmd.STOP);
6    }
7    public void compute() {
8      if (this.state.equals(State.idle)
9            && (this.bump.getCurrentValue() != null
10           && this.bump.getCurrentValue() == true) ) {
11       this.rightMotor.setNextValue(MotorCmd.FORWARD);
12       this.leftMotor.setNextValue(MotorCmd.FORWARD);
13       this.state = State.driving;
14     }
15     // ... more transitions here
16   }
```

Listing 4: An excerpt from the Java code generated from the automaton of BumpControl.

3 MontiCore Language Framework

We have developed the ADL MontiArc [HRR12] and the corresponding framework using the language workbench MontiCore [KRV10]. MontiCore facilitates the development of domain specific modeling languages by providing a grammar for language definition

and tools for parser generation, symbol table management, code generation and a context conditions framework. MontiCore languages like MontiArcAutomaton, are defined by context-free grammars. To check properties not expressible in context-free grammars, e.g., whether a variable is defined twice, MontiCore provides a compositional Java-based context condition framework [Vö11]. As these context conditions often require information from other models (e.g., to determine whether an assignment violates a type constraint), MontiCore also contains a compositional symbol table framework [Vö11], to facilitate development of complex context conditions.

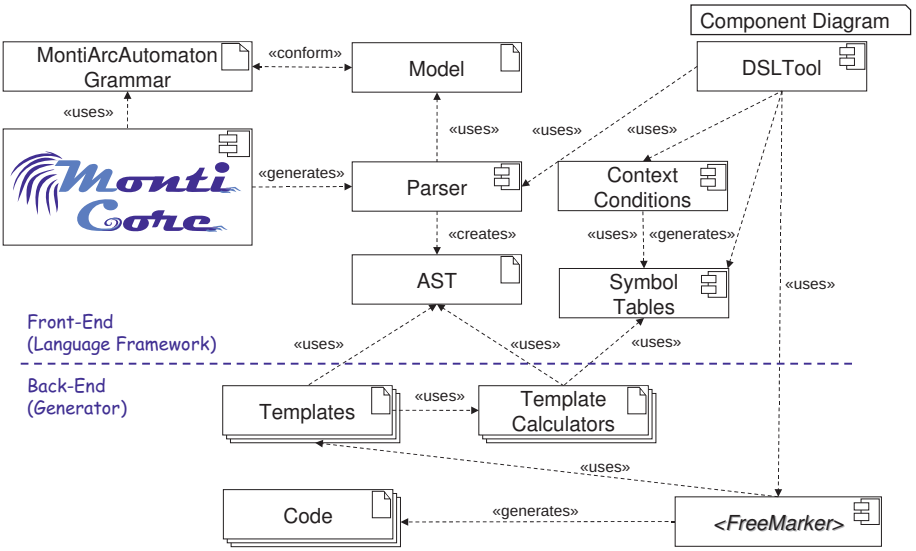


Figure 5: MontiCore uses the grammar to generate a parser for MontiArcAutomaton models which creates the AST (see [Sch12]). The `DSLTool` uses the parser to read models, which are validated by the context condition framework using the symbol tables provided by the `DSLTool`. The `DSLTool` further may use FreeMarker templates and template calculators to generate code from the models based on the AST.

The component diagram in Figure 5 illustrates the components of the MontiCore DSL framework: the compositional approach of MontiCore facilitated development of MontiArcAutomaton by generating a parser for the MontiArcAutomaton DSL and providing frameworks for context conditions, symbol table generation and code generation. The code generation framework of MontiCore [Sch12] utilizes the Java-based template engine FreeMarker[4] to construct new code generators. Using this framework, a new code generator usually only requires a few new calculators and templates. MontiCore further facilitates language and generator development by means of language inheritance and composition mechanisms (like the embedding of I/O$^{\omega}$ automata into MontiArc). We discuss these, their influence on the code generation, and the reuse of calculators and templates in Section 5.

MontiCore also provides a framework for the generation of text editor Eclipse plugins.

[4]Freemarker Template Engine: `http://freemarker.org/`

This framework offers a DSL which allows to define editor models and a set of workflows which editor developers have to implement for their languages. Using this and the Ecore generation of MontiArcAutomaton, we have developed a both textual and graphical editor for MontiArcAutomaton models as displayed in Figure 6.

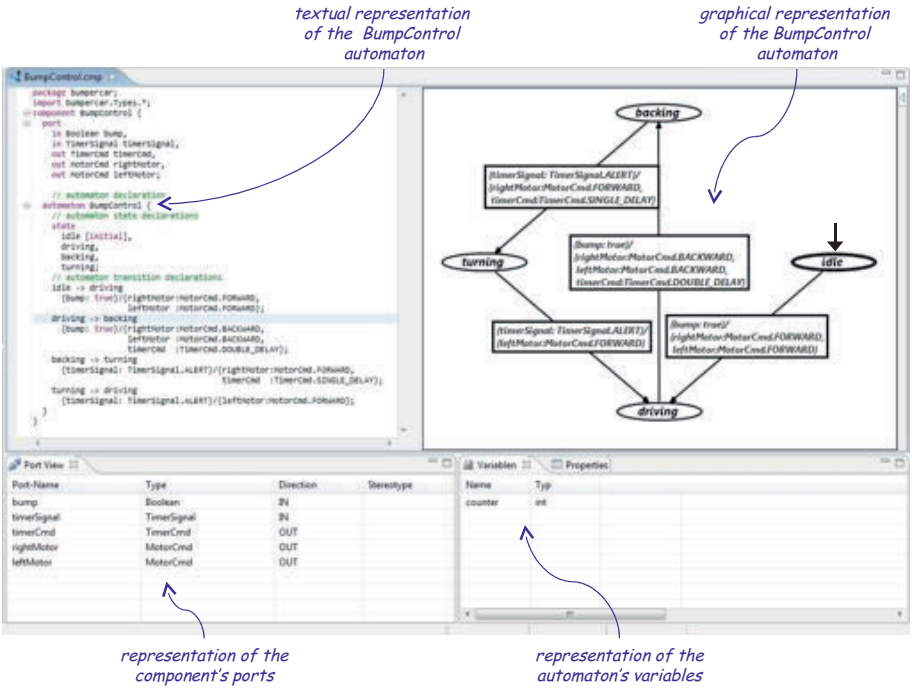


Figure 6: The combined textual and graphical editor for MontiArcAutomaton models featuring parallel textual and graphical editing of the same model.

For graphical editing, we have defined a generic automaton language using the EMF Ecore meta model. MontiArcAutomaton models are generated into models of a generic automaton language and displayed using the Eclipse Graphical Editing Framework[5]. We chose not to define a distinct I/O$^{\omega}$ automaton EMF language, as we want to embed other kinds of automata into MontiArc components in the future and thus would have to define new EMF Ecore languages for each kind.

[5]Eclipse Graphical Editing Framework: `http://www.eclipse.org/gef/`

4 MontiArcAutomaton: a Software Architecture Structure and Behavior Modeling Language

We use the ADL MontiArc [HRR12] to model CPS. MontiArc describes architectures using components and connectors. Components encapsulate subsets of the systems functionality and regulate control via explicitly defined interfaces, which in MontiArc are sets of directed typed ports. The type of a port determines the possible messages a component may receive or send on that port, and can be defined using UML/P class diagrams [Rum11]. Connectors effect and regulate the interaction of components.

MontiArc realizes the semantics of FOCUS [BS01] to describe component behavior. Components are treated as black-boxes that read input streams and produce output streams (message streams). These streams are the observable history of component interactions. A MontiArc component is either *atomic* and its behavior is defined explicitly or *composed* and its behavior is defined solely by the structural composition of the behaviors of its subcomponents. Models of composed components define the relations between subcomponents and connectors. MontiArc distinguishes between the definition of a component and its instantiation, supports powerful typing, instantiation, and parametrization mechanisms as described in [HRR12], but does itself not provide a language to model the behavior of its components explicitly. We thus have extended MontiArc by embedding I/O$^{\omega}$ [Rum96] as component definitions. The semantics of I/O$^{\omega}$ automata are stream processing functions. Our MontiCore implementation of this language is called MontiArcAutomaton.

4.1 The MontiArcAutomaton Language

We added local variables, states and transitions to MontiArc components to model component behavior. Variables can be used by automata to store and look up intermediate values. A transition connects a source state with a target state and has an optional guard, input block, and output block. Variables, states and transitions are only visible inside a component and all communication between components is made explicit via ports and connectors. I/O$^{\omega}$ automata do not feature hierarchical states because decomposition takes place on component level and thus reduces the complexity of component behavior description significantly. The input block of a transition defines patterns of messages and events received on incoming ports or stored in the local variables of the component that together with the guard activate the transition. A guard is a predicate over the messages on input ports and values stored in local variables. Guards in MontiArcAutomaton can be specified using OCL/P [Rum11, Sch12] a MontiCore implementation of OCL. The reaction of a component to an input is specified by the output block, which is a set of pairs of output ports and the (streams of) messages that are sent as a reaction to the input. It also may contain assignments to the local variables of a component.

Listing 7 displays an excerpt from the concrete syntax of the automaton of component `BumpControl` (see Figure 2). The state `idle` is an initial state and defines initial outputs

MontiArcAutomaton

```
1  component BumpControl {
2    // Port declarations
3    automaton {
4      state
5        idle [initial {rMotor:STOP,
6                       lMotor:STOP}],
7        driving, backing, turning;
8
9      idle -> driving {bump:true} / {rMotor:FORWARD,
10                                    lMotor:FORWARD};
11     // ... more transitions here
12   }}
```

Listing 7: An excerpt from the concrete syntax of the automaton of component `BumpControl`.

in ll. 5-6. The transition from state `idle` to state `driving` is shown in ll. 9-10. It only reads from one port but defines the output on multiple ports. In general, transitions may read from all incoming ports and send messages on all outgoing ports.

The semantics of MontiArcAutomaton is described as sets of stream processing functions (SPF) [Rum96, RR11]. In the case of a total and deterministic automaton the set is a singleton, otherwise each stream processing function (SPF) describes a different possible implementation of the desired system. The SPF corresponding to a MontiArc component maps one input stream bundle to one output stream bundle. The input stream bundle contains streams for each input port of the component and the output stream bundle contains streams for each output port of the component. For a formal definition of MontiArcAutomaton semantics see [Rum96] and the language report on the MontiArcAutomaton website [Mon12].

5 Code Generation from MontiArcAutomaton

The code generation framework of MontiCore uses FreeMarker templates and adds *template operators* and *template calculators* to transform the ASTs of models of MontiCore languages into implementations. Template operators provide the infrastructure for code generation, access the AST, call template calculators and include sub templates, e.g., for connectors or transitions. Templates consist of target language fragments and FreeMarker control structures. Template calculators perform complex operations and provide symbol table access that would complicate the templates or be impossible inside templates due to FreeMarker restrictions. The template operator also persists the results of calculations.

Listing 8 illustrates these concepts on the template for a single transition in Python. The template executes the method `getFrom()` on the current AST node (l. 1). Line 2 uses the FreeMarker directive `<#if>..</#if>` to determine and evaluate the transition's guard calling the template calculator `guardCalculator`. The result of this evaluation is stored in the field `guardExpression` of the `guardCalculator` and thus available

in the template via the template operator. Afterwards, the template iterates over all input ports/channels (l. 5) and includes a template called `concatStream` for each input (l. 8).

FreeMarker

```
1  if (self._state == ${op.getValue("enumName")}.${ast.getFrom()}
2  <#if op.callCalculator(guardCalculator)>
3    and (${op.getValue("guardExpression")})
4  </#if>
5  <#foreach chin in ast.getChannel_in()>
6    and (self._${chin.getInName()}.getCurrentValue() != None
7          and self._${chin.getInName()}.getCurrentValue() ==
8            ${op.includeTemplates(concatStream, chin.getInput())})
9  </#foreach>
10   ):
11 <#foreach chout in ast.getChannel_out()>
12   self._${chout.getOutName()}.setNextValue(
13     ${op.includeTemplates(concatStream, chout.getOutput())})
14 </#foreach>
15   self._state = ${op.getValue("stateEnumName")}.${ast.getTo()}
```

Listing 8: The template for the Python implementation of a single transition in FreeMarker.

Parts of the Python code generated from the automaton of component `BumperControl` (from Figure 2) are shown in Listing 9. The initial state and output of the component are set in ll. 4-6. The code of the transition from state `idle` to `driving` (shown in ll. 8-14) is generated based on the template from Listing 8.

Python

```
1  class BumpControl(runtime.Component):
2
3    def init(self):
4      self._state = BumpControlState.idle
5      self._rMotor.setCurrentValue(MotorCmd.STOP)
6      self._lMotor.setCurrentValue(MotorCmd.STOP)
7
8    def compute(self):
9      if (self._state == BumpControlState.idle and
10         (self._bump.getCurrentValue() != None and
11          self._bump.getCurrentValue() == True) ):
12       self._rMotor.setNextValue(MotorCmd.FORWARD)
13       self._lMotor.setNextValue(MotorCmd.FORWARD)
14       self._state = BumpControlState.driving
15       # ... more transitions here
```

Listing 9: An excerpt from the Python class generated from the automaton inside component `BumpControl` (cf. Lst. 3 and 4)

Using this framework, we have developed four code generators by implementing new templates and reusing most of the MontiArcAutomaton dependent template calculators.

The next section explains how MontiCore supports reuse in code generator development.

5.1 Multiple Target Language Code Generation

MontiCore facilitates toolchain reusability as grammar, symbol table, and context conditions are part of the language front-end. They can easily be reused for every new code generator and editor. Figure 10 illustrates this reuse. Common workflows and code shared among the different back-ends (code generators) for MontiArcAutomaton, are packaged in the project `MontiArcAutomatonBECommons`. This way, code generation reuses calculators common to all back-ends (e.g., collection of states and transition) as well as symbol tables entries and context conditions generated and implemented for the different inherited and embedded modeling languages respectively.

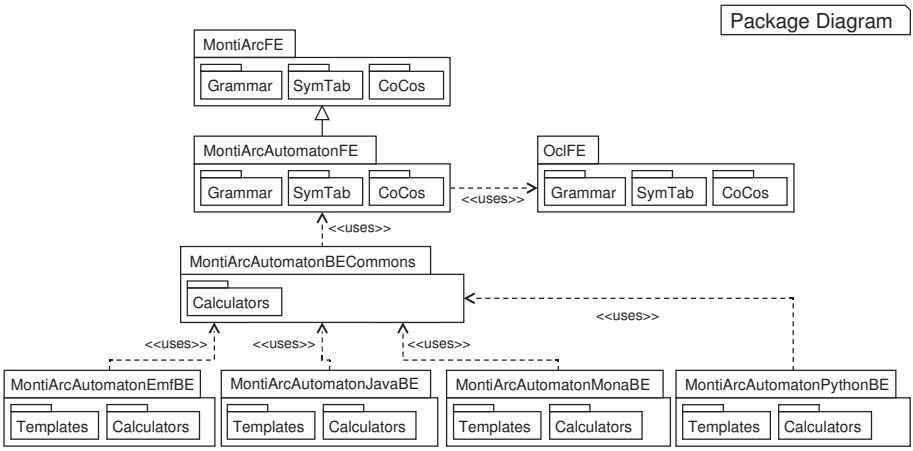


Figure 10: Dependencies between language front-ends (containing languages and context conditions) and back-ends (containing generators).

We have developed (code) generators from MontiArcAutomaton models to EMF Ecore (`MontiArcAutomatonEmfBE`), Java code (`MontiArcAutomatonJavaBE`), Mona (`MontiArcAutomatonMonaBE`) and Python (`MontiArcAutomatonPythonBE`). These generators reuse the parsers, symbol table generation, and context conditions of MontiArc, OCL/P, and MontiArcAutomaton. Overall, for these four code generators, we developed between one (EMF) and 3 (Mona) target language specific templates as well as between two (EMF) and five (Python) target language specific template calculators: as Python, for example, lacks explicit types, a new calculator for the parameter lists of method signatures had to be developed. The average template lengths – including comments – range from eleven lines (Mona) to 48 lines (EMF). Table 11 quantifies this reuse. As all generators use similar structures (e.g., iteration over the automaton's transitions), we could identify and reuse seven common template calculators (provided by `MontiArcAutomatonBECommons`).

Generator	Number of (additional) Calculators	Number of Templates	Average Template Length
MontiArcAutomatonBECommons	7	-	-
MontiArcAutomatonEMFBE	1	2	48
MontiArcAutomatonTSBE	2	12	33
MontiArcAutomatonMonaBE	3	11	11
MontiArcAutomatonPythonBE	2	13	29

Figure 11: Key figures on calculator reuse in code generator development. The generator project `MontiArcAutomatonBECommons` contains generic MontiArcAutomaton calculators. The other projects add additional target platform specific calculators.

Also, the target languages pose some restrictions on valid models of the input language. For example, the code generation to Java supports generic port types not supported by the Mona code generation. As another example, the analysis using Mona heavily depends on underspecification mechanisms, e.g., undefined inputs and outputs and non-deterministic transitions. The Java and Python code generators do not translate these models properly. Thus, we have developed sets of context conditions that define language profiles of MontiArcAutomaton for different generators.

Besides toolchain reusability, MontiCore also facilitates language reusability as these may inherit from others (MontiArcAutomaton inherits from MontiArc) and embed other languages (MontiArcAutomaton embeds OCL/P). The new language can reuse context conditions, template calculators, and symbol tables from the inheriting or embedding language.

We want MontiArcAutomaton models to be platform independent: this no longer works, when dealing with hardware or API access. We therefore postulate the existence of (and have developed) a runtime library per target platform. This library contains component models that deal with target system interaction (e.g., hardware access, usage of other frameworks, etc.). For the robotics project, we implemented such a library consisting of components wrapping access to sensors and motors, such that the system engineers were able to model the systems without taking the underlying API into account.

5.2 Challenges: Equal Operational Semantics and Libraries

Equal Operational Semantics We have developed multiple code generators for the same language. During system development a single model is used with the EMF Ecore code generator for editing, the Mona code generator for validation and verification and the Java LeJOS as well as the Python ROS code generators for execution on different platforms. For the latter three translations it is of great importance that all models behave according to a single operational semantics (to the extend necessary to preserve properties established in prior verification and validation steps).

Enforcing a common operational semantics mutually affected the code generators. On

one hand, the common operational semantics requires synchronization of communication and computation on distributed CPUs in our Java and Python implementations, since our implementation in Mona only supports synchronized components (the current translation can not model unbounded buffers). On the other hand, it also requires the modeling of null values on message streams in the Mona implementation. Furthermore, we had to choose a compatible encoding of messages and of communication protocols (partially implemented using hardware wrappers) between robots running LeJOS or ROS.

In the code generators presented here we have addressed the challenge of ensuring equal operational semantics by careful inspection of the code generators (facilitated by the common structure) and validation based on test cases.

Libraries Model and code libraries are important in every language to provide reuse based on common interfaces. The MontiCore symbol table framework [Vö11] implements a conform look-up and handling of components independently of their source (library or model) and implementation (generated or manual). All tools operating on the model level, e.g., the content assistants of our editors and context condition checks, thus fully support library components.

For our code generators pure model libraries are no problem since the complete implementation can be generated for any target language. As discussed in the previous section, we also employ library components that need target platform specific implementations we can not describe with the MontiArcAutomaton language. We never the less model these library components as MontiArcAutomaton models to provide consistent interfaces on the model level. In addition, we have implemented mechanisms to combine the generated code with manual implementations without modifying the generated code (e.g., the factory pattern and delegation). Since platform specific implementations, e.g., hardware access, are not contained in the models, this code has to be manually created for every target platform. The realization of components corresponding to hardware wrappers is still a challenging manual task in our Mona implementation.

6 Related Work

The AutoFOCUS [HST10, HF07] tool chain for model-based development of reactive, distributed systems supports modeling of logical architectures (similar to MontiArc), technical/platform architectures and deployment mappings. MontiArcAutomaton and AutoFOCUS share the same semantic domain FOCUS [BS01]. Behavior of AutoFOCUS components can be modeled using input/output automata similar to MontiArcAutomaton [HF07]. The AutoFOCUS tool chain contains code generators to multiple target platforms [HST10], e.g., to C code and to the theorem prover Isabelle/HOL. To the best of our knowledge the AutoFOCUS verification mechanisms do not support fully automated refinement checking as shown in [Kir11] and code generation to different robotics platforms as presented in this paper.

Modeling Languages The UML [OMG12] is a general purpose modeling language family consisting of 14 diagrams for structure, behavior and interaction modeling. Among

these are *state machine diagrams*, to model the behavior of systems in terms of states and transitions, and *component diagrams*, to model the interaction of components via the ports of their interfaces. While these could be used to model architecture and behavior of CPS, the semantics of the embedding of state machines into components is not explicitly defined in UML.

SysML [FMS11] is a general-purpose modeling language based on UML. The language features several diagram types known from UML and introduces additional ones. Among these diagrams are UML *state machine diagrams* and *internal block diagrams* based on UML *composite structure diagrams*. While the latter, provided to model the internal structure of classes using blocks and ports, might be used to model a system architecture, the UML state machine diagrams still suffer the problems mentioned above.

Language Workbenches The MontiCore approach to language design is similar to the Eclipse Xtext [EB10] approach, which also uses EBNF-like grammars to generate parser, text editor eclipse plugin, and further tooling. The AST generated from Xtext parsers is EMF-based and code generation uses the Eclipse Xtend programming language[6] to define the templates used for code generation. Xtend augments Java with several concepts while compiling into interoperable Java source code. As Xtext requires Eclipse, the integration into tool chains is hampered. MontiCore offers Eclipse integration and in addition API and command line tools that are independent of the Eclipse framework.

The Meta Programming System (MPS)[7] follows a different approach, as it follows a projectional approach, i.e., lets the user directly edit a model's underlying AST. MPS uses a proprietary meta model and does neither provide Eclipse integration, nor means of automated toolchain integration. The code generation with MPS is implemented as a model-to-model transformation, which requires grammars of all target languages. Our approach does not require grammars of the target languages.

7 Conclusion

We have presented the MontiArcAutomaton modeling language to model architecture and behavior of CPS and illustrated how the development of CPS can be improved using modeling and verification techniques. The MontiArcAutomaton framework contains four code generators from platform independent MontiArcAutomaton models to Mona (formal analysis), EMF Ecore (graphical editing), and Java and Python (deployment).

Based on these code generators we have illustrated how the MontiCore DSL framework supports development and reuse of code generators, including (partial) reuse of symbol tables, context conditions, and template calculators.

Currently, we are working on a modeling language to specify the deployment of MontiArcAutomaton components to hardware. Furthermore, we are conducting an evaluation of the MontiArcAutomaton framework with master students.

[6] Xtend programming language: `http://www.eclipse.org/xtend/`
[7] Meta Programming System: `http://www.jetbrains.com/mps/`

References

[BBC+07] Davide Brugali, Alex Brooks, Anthony Cowley, Carle Côté, Antonio Domínguez-Brito, Dominic Létourneau, Françis Michaud, and Christian Schlegel. Trends in Component-Based Robotics. In *Software Engineering for Experimental Robotics*, volume 30 of *STAR*, chapter 8, pages 135–142. Springer, Berlin, Heidelberg, 2007.

[BR12] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Proceedings of the INFORMATIK 2012*, 2012.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[EB10] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Computer-Aided Verification, (CAV '98)*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, 1998.

[FMS11] Sanford. Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. The MK/OMG Press. Elsevier Science, 2011.

[FR07] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE.*, pages 37–54, 2007.

[GBWK09] Michael Geisinger, Simon Barner, Martin Wojtczyk, and Alois Knoll. A software architecture for model-based programming of robot systems. *Advances in Robotics Research*, pages 135–146, 2009.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HF07] Florian Hölzl and Martin Feilkas. AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *MBEERTS*, volume 6100 of *LNCS*, pages 317–322. Springer, 2007.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen, february 2012.

[HST10] Florian Hoelzl, Maria Spichkova, and David Trachtenherz. AutoFocus Tool Chain. Technical Report TUM-I1021, TU Munich, 2010.

[Kir11] Dennis Kirch. Analysis of Behavioral Specifications for Distributed Interactive Systems with MONA. Bachelor Thesis, RWTH Aachen University, 2011.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: a framework for compositional development of domain specific languages. *STTT*, 12(5):353–372, 2010.

[Lee06] Edward A. Lee. Cyber-Physical Systems - Are Computing Foundations Adequate? In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, 2006.

[Mon12] MontiArcAutomaton website and language report. http://www.se-rwth.de/materials/ioomega, 2012. Accessed 12/2012.

[NFBL10] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. *Design Principles of the Component-Based Robot Software Framework Fawkes*, volume 6472 of *LNCS*, chapter NFB+10, pages 300–311. Springer, Darmstadt, Germany, 2010.

[OMG12] Object Management Group. Unified Modeling Language: Superstructure Version 2.4.1. http://www.omg.org/spec/UML/2.4.1/, 2012. Accessed 10/12.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 5(1-2):29–53, July 2011.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In *Modelling and Quality in Requirements Engineering*, pages 133–146. Monsenstein und Vannerdat Münster, 2012.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Xpert.press. Springer Berlin, 2nd edition, September 2011.

[Sch09] Bernhard Schätz. *Model-Based Development of Software Systems: From Models to Tools*. Habilitation thesis, TU Munich, 2009.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SSL11] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model. In *Introduction to Modern Robotics*. iConcept Press, 2011.

[TL08] Sebastian Thrun and John J. Leonard. Simultaneous Localization and Mapping. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 871–889. Springer, 2008.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons, Inc., 1 edition, 2009.

[Vö11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011.

Paisley: A Pattern Matching Library for Arbitrary Object Models

Baltasar Trancón y Widemann
Technische Universität Ilmenau
baltasar.trancon@tu-ilmenau.de

Markus Lepper
<semantics/> GmbH

Abstract: Professional development of software dealing with structured models requires more systematic approach and semantic foundation than standard practice in general-purpose programming languages affords. One remedy is to integrate techniques from other programming paradigms, as seamless as possible and without forcing programmers to leave their comfort zone. Here we present a tool for the implementation of *pattern matching* as fundamental means of automated data extraction from models of arbitrary shape and complexity in a general-purpose programming language. The interface is simple but, thanks to elaborate and rigorous design, is also light-weight, portable, non-invasive, type-safe, modular and extensible. It is compatible with object-oriented data abstraction and has full support for nondeterminism by backtracking. The tool comes as a library consisting of two levels: elementary pattern algebra (generic, highly reusable) and pattern bindings for particular data models (specific, fairly reusable, user-definable). Applications use the library code in a small number of idiomatic ways, making pattern-matching code declarative in style, easily writable, readable and maintainable. Library and idiom together form a tightly embedded domain-specific language; no extension of the host language is required. The current implementation is in Java, but assumes only standard object-oriented features, and can hence be ported to other mainstream languages.

1 Introduction

Declarative (functional or logical) languages are more or less equally powerful both when creating compound data, and when extracting their components: Term/pattern constructors on the right/left hand side of definition equations, respectively, offer balanced syntactic support with clean, inverse algebraic semantics. Object-oriented languages, by contrast, mostly offer the desirable feature of *compositionality* only when creating objects, but lack a corresponding primitive idiom for extraction. Instead, explicit getter methods, type casts, assignments to local variables and explicit case distinction have to be applied in an imperative programming style, which is inadequate to the purpose of mere querying.

Obviously it is desirable to enrich object-oriented programming practice by techniques from more declarative styles, together with the corresponding supporting infrastructure. "Declarative" in this context means that access operations form an algebra and entail semantic properties by induction in their structure. If this is done in a smooth and natural way, it will make program source texts more efficient and enjoyable to write, as well as better readable and maintainable.

There are techniques, like the *visitor* and *rewriter* patterns, which introduce a more declarative style of writing in object-oriented data evaluation. In [LT11], we have demonstrated how visitor-based extraction can be optimized using a combination of static and dynamic analyses. However, this technique corresponds to a more global, "point-free" way of formulating queries and is too heavy-weight and semantically too loosely defined for the purpose of point-wise extraction of details, for which local access operations *are known*.

Here, we investigate the import of a concept well-proven in all kinds of programming styles into the object oriented paradigm, namely *pattern matching*: The Paisley library presented below is a generic programming aid for data extraction by pattern matching that unifies desirable features of declarative paradigms with a pure object-oriented approach to data abstraction. It comes in two parts: a basic library and a programming idiom that uses the library operations as its core vocabulary. Problem-specific composite operations can be provided by the user by extending the library cleanly through subclassing. Our implementation is hosted in Java, but nothing prevents the same technique to be used in other strongly typed object-oriented environments such as C++ or .NET.

The present article extends an earlier tool demonstration paper [TL12] with more recent features, additional technical details and a comparative evaluation of the design. Detailed API documentation and downloads are available online at [TL11].

2 Standards of Pattern Matching

Pattern matching, in the wide sense, plays an important role in many different kinds of programming environments. But a close look shows that the techniques applied in the various fields differ substantially regarding theoretical foundation and expressiveness, the treatment of nondeterminism, type discipline, etc. These are the relevant role models, positive or negative, for our approach:

String Processing with Regular Expressions Here typing is a trivial matter, since patterns refer to character strings only. Theoretical foundation is sound; recently sound semantics have been defined even for backward group references [BC08]. Nondeterminism is either resolved locally by various flavours (greedy, reluctant etc.) of operators, or exposed to the user as global search. Focus is often on performance-critical applications, such as real-time filtering of high-frequency network traffic, making compilation to specially designed automata the technique of choice. Consequently, object-oriented data models play no role in those scenarios, and our approach does not apply.

Functional Programming with Algebraic Datatypes Inverse constructors are central to data extraction and equational function definition in functional programming languages (Hope, ML, Haskell, etc.), and share the full type discipline of the language. Nondeterminism arises not within one pattern, but rather between overlapping patterns of equations, and is usually resolved implicitly by a first-fit rule. In the context of the multi-paradigm language Scala, pattern matching has moved closer to object-oriented programming, by virtue of the `case class` construct and the `unapply` magic method; see [EOW07].

XML Navigation and Deconstruction For this purpose the XPath [CD99] pattern notation is the basis for the majority of transformation systems, like XSL-T, Xquery, XJ, Xact, JDOM and more. There is a complete theory for a subset of XPath excluding data value comparisons [GL06]. Alternative formalisms are less popular but extant; e.g. XDuce [HP00] is a functional language with patterns serving as types, with full static checking, and with regular expressions over patterns to match heterogeneous lists.

Logic Programming with Goals and Unification Logic programming languages (Prolog etc.) offer a distinct quality by making nondeterminism, unification of terms with free variables, and exhaustion of solution spaces (encapsulated search) first-class constructs of the language. They are usually weakly typed, but theoretically well explored.

Model Query and Transformation In dedicated model query languages pattern matching is a central functionality as well: the evaluation of a query delivers a subset of model nodes. Selection criteria range from simple checks on attribute values to complex relational constraints. In graph transformation systems, graph patterns feature prominently as the left hand sides of rewrite rules. The pervasive nondeterminism in graphs is often resolved by explicit control flow. See for instance the "Rule Application Control Language" of GrGen.NET [BGJ11]. Pattern notations take a vast number of theoretically and pragmatically distinct forms in the multitude of existing systems. For instance, the query language GReQL [EB10] offers regular path expressions to express complex patterns.

3 Design of Paisley Pattern Matching

3.1 Requirements

Porting pattern matching to an object-oriented environment is not a trivial task. On one hand, there are semantic problems to be solved, mostly pertaining to the impedance mismatch between object interfaces and algebraic pattern calculi. On the other hand, there is a multitude of theoretically possible implementation techniques. The Scala paper [EOW07] gives a good survey on different strategies, complemented with experimental evaluation. At the end of this article we will apply their criteria to our solution.

The Paisley approach is distinguished by a carefully selected canon of rigorous design requirements:

1. *Declarative, readable, writable, customizable.* Patterns express the programmer's intention of data extraction with as little formal noise as possible. This improves significantly over standard imperative/object-oriented patterns in terms of self-documentation and maintainability.
2. *Full reification: no parsing/compilation overhead at runtime.* Patterns are typed host-language objects; ill-defined usage is detected at compile time. This makes our approach diametrically opposed to dynamic notations, in particular traditional regular expression libraries such as `java.util.regex`.

3. *Statically type-safe variables.* No need to down-cast variable bindings or check their types at runtime.

4. *Statically type-safe patterns.* Detect ill-typed pattern matching attempts as often as possible.

5. *No language extension: independent of host compiler/VM.* Solution can be used transparently with off-the-shelf programming platforms and runtime environments.

6. *No assumptions on host language beyond standard OOP.* Solution can be reimplemented in any standard object oriented programming language. Custom extensions can use the full power of the host language, at the user's discretion and risk.

7. *No adaptation of model datatypes required.* Applies equally to data models from third-party repositories or developed without pattern matching in mind; no source access required.

8. *Support for multiple views per type.* Different collections of patterns can expose different structural aspects of a data model. Sharpens the preceding requirement.

9. *Support for continuation-style nondeterminism.* Patterns are ordinary objects with hidden inner state which *locally* and completely memorizes the current backtracking situation. Access to successive matches should be postponable indefinitely, even across serialization and de-serialization of all objects involved.

10. *Nondeterminism incurs no significant cost unless actually used.* Implies absence of central storage or control mechanisms, and lazy exploration of alternatives.

From the programming language perspective, the main focus is on *strict typing*. This is enforced by type relations of different kinds which are mapped to the type system of the host language, and thus inherit its checking and diagnostics facilities (for API see Fig.1):

1. *Pattern and data.* The type of data which can be matched against a given pattern is described by a parameter of the patterns' type: An instance of class `Pattern<A>` will match all instances of type `A`.

2. *Pattern lifting, contravariantly.* The type of any function which lifts a pattern on an object's field to a pattern on the object as a whole, or from a member object to a collection, etc., is always a function type between the corresponding pattern types: An access operation on class `A` that yields a subobject of type `B` induces a lifting function from `Pattern<B>` to `Pattern<A>`.

3. *Pattern combinators respect data types.* The Paisley pattern combinators require compatible types of the patterns' targets: A `Pattern<A>` and a `Pattern<B>` combined always result in a `Pattern<C>` where `C` is a subtype of both `A` and `B`.

4. *Pattern variables limit the type of their possible results.* On the construction side, a pattern variable has a type attributed with the type it can match, as any other pattern. After successful match, on the binding side, the variable offers a typed

getter interface: A `Variable<A>` is a `Pattern<A>` and yields values of static type `A`.

3.2 Basic Implementation Technique: DSL by Library + API

Pattern matching directives can be seen as a *domain specific language* (DSL) to be embedded into a general-purpose host programming language, in this case Java. For this there exist some well-known basic philosophies:

The requirements (2)–(4), for static type safety and reification rule out mere textual encodings, as criticized above. On the other hand, the requirement (5) for compiler independence rules out implicit compile-time handling of pattern matching code. Another possible solution is a *generative* approach, where DSL front-end syntax is translated into host language source code in a dedicated pre-processing step. This approach is used by many of the authors' other tools.

Here we chose instead an API and library-based implementation: Patterns are constructed at run-time, in terms of host language objects with certain *semantics*. We prefer this approach because it is far more lightweight and flexible. Of course, if appropriate, complex stereotypical code fragments on top of this library can be generated automatically from a more concise domain-specific notation, as for instance done by our umod tool [LT11].

3.3 Imperative View on Pattern Matching

The classical semantics of patterns as the inverse of constructor terms of algebraic datatypes, de-facto standard in declarative languages, does not carry over smoothly to the object-oriented paradigm, because object constructors generally lack the mathematical benevolent properties of their algebraic counterparts, namely extensionality, injectivity, disjointness and completeness.

A looser notion of pattern matching, more appropriate to the abstraction style of object orientation, is to consider it the reification and composition of certain categories of data extraction operations: *Testing* classifies objects as either acceptable or not; *projection* descends into the structure of the subobjects and extracts primitive data attributes; *binding* assigns data to variables.

These three aspects can be delimited precisely in well-written object-oriented code; they correspond to simple local idioms. A fourth aspect however, namely *logic*, is implicit and scattered across the control flow structure of code, in terms of sequences, conditionals, case distinctions, loops, etc. That the logical aspect comes with subtle and non-local ramifications should be evident to everyone who has successfully hand-coded a parser. This is a major source of difficulties in writing, reading and maintaining object-oriented code. Our central motivation behind the Paisley approach is to put logic on equals footing with the former three aspects in an object-oriented setting.

```
abstract class Pattern<A> {
  public abstract boolean match(A target);
  public boolean matchAgain();

  public static <A> Pattern<A>
    both(Pattern<? super A> first, Pattern<? super A> second);
  public static <A> Pattern<A>
    either(Pattern<? super A> first, Pattern<? super A> second);
}

class Variable<A> extends Pattern<A> {
  public A getValue();

  public <B> List<A>     eagerBindings(Pattern<? super B> root, B target);
  public <B> Iterable<A> lazyBindings(Pattern<? super B> root, B target);

  public <B> Pattern<B> bind(Pattern<? super B> root, Pattern<? super A> sub);
  public Pattern<A> star(Pattern<? super A> root);
  public Pattern<A> plus(Pattern<? super A> root);
}

abstract class Transform<A, B> extends Pattern<A> {
  protected final Pattern<? super B> body;

  protected abstract B apply(A target);
  protected abstract boolean isApplicable(A target);
}
```

Figure 1: Interface synopsis (core)

The design of our library is such that these four concerns are separated as much as possible, but can be composed as freely as required. A notable implication is that logical structure, in particular with respect to nondeterminism, is given the most fundamental operator basis possible, namely fully compositional ad-hoc conjunction and disjunction of subpatterns, of which traditional pattern aggregation and case distinction are merely special cases.

3.4 The `Pattern` interface

The main interface of the library is the abstract base class `Pattern<A>` of patterns that can process objects of type `A`. A pattern `Pattern<A> p` is applied to some target data `x` of type `A` or any subtype by calling `p.match(x)`, returning a Boolean value indicating whether the match was successful.

In case the result is **`true`**, variables occurring in the pattern are guaranteed to be bound under conditions inductive in the logical structure of the pattern: A successful match binds variables in all branches of a conjunction, and in some branch of disjunction. Conversely, a variable is certainly bound if it occurs in all disjunctive branches or in some conjunctive branch. In case the matching result is **`false`**, variable bindings are unspecified.

After matching successfully, and using the values of bound variables, the parameter-less method `p.matchAgain()` may be called. This is how nondeterminism, that is the fact that a given pattern matches a given target *in more than one way*, is exposed at the interface.

The call of `matchAgain()` causes a new matching attempt of the same target by backtracking. The result has the same interpretation as for `match(x)`, so `matchAgain()` can be iterated as long as its result is **`true`**. The match is different in some way, in the sense the some new disjunctive branch is taken, in each successful call. The default implementation of `matchAgain()` always returns **`false`**, specifying a deterministic pattern.

Iteration over all possible matches of a nondeterministic pattern is effected simply by a **`do`** ... **`while`** loop, with minimal redundancy:

```
if (p.match(x)) do
  doSomething();
while (p.matchAgain());
```

3.5 Predefined Tests and Combinators

The Paisley library offers factory methods for patterns wrapping ubiquitous test and getter methods, and generic pattern combinators and liftings.

Basic rule for the whole implementation is strict typing, as postulated above in section 3.1. In this context it is essential to observe that all patterns except variables are *contravariant*: A pattern capable of matching any supertype `B` of `A` can act as a `Pattern<A>`, hence `Pattern<B>` should be treated as a subtype. This is expressed by library methods consistently taking parameters of wildcard types with lower bounds, in forms such as `Pattern<? super A>`.

In the current implementation there are static factory classes `ReflectionPatterns`, lifting some Java reflection operations such as `isInstance` or `getAnnotation`, as well as `StringPatterns` which lifts standard string operations like `startsWith`, but also the interface of the `java.util.regex` package. `PrimitivePatterns` wraps Java primitive types and some core methods such as `equals` or `compareTo`.

`CollectionPatterns` lifts patterns on elements to patterns on collections. This can be used as a controlled source of nondeterminism: Search patterns such as constructed by `anyElement(Pattern)` try all contained elements for `match()`/`matchAgain()`, while deterministic patterns such as constructed by `get(int, Pattern)` try to match only the one element at the given position. Variants for array types are also provided.

The class `Pattern` itself provides a framework of logical core combinators: binary operators `both` for conjunction and `either` for disjunction, to be discussed in detail in section 3.8 below; the constant patterns `any()` and `none()`, matching everything and nothing, respectively, as base cases; n-ary vararg combinator variants for convenience.

Modifications of the solution space of patterns are implemented as instance methods. `p.noMatch()` yields a pattern that matches if `p` itself has no solution. The match is deterministic and binds no variables; compare to negation-by-failure in logic programming. `p.uniquely()` matches iff `p` itself matches with exactly one solution, that is `p.matchAgain()` fails immediately. The match binds all variable also bound by `p`.

3.6 Variables

A pattern variable is simply a pattern of class `Variable<A>` that matches always, and binds to the matched object for later retrieval via the `getValue()` method. The variable interface is unique in the sense that its type parameter occurs in a return type, so it does not behave contravariantly as other pattern constructs do; cf. the preceding section 3.5.

The basic idiom of pattern matching is thus:

```
Variable<C> vc = new Variable<C>();
Variable<D> vd = new Variable<D>();
Pattern<A> p = myPattern(vc, vd);   // known to bind vc AND vd
if (p.match(x))
  doSomething(vc.getValue(), vd.getValue());
```

It is not by accident that the pattern variables `vc` and `vd` in this example have local declarations with precise static type (first two lines): This style enables the full use of static type information for bound values, even if the matching pattern has been constructed from generic building blocks that are defined independently of the type of occurring variables.

Figure 2 shows how variables are used in a Paisley compound pattern:

References to variables must be retained explicitly; they are not accessible via the containing pattern, as this would break compositionality. Therefore they must be constructed first, retaining a reference, before being incorporated into a newly constructed pattern. In order to safeguard against race conditions, it is good practice to give pattern variables local visibility only.

Variables are in the imperative style, that is simple, mutable containers for a single value; they do not provide either the unification functionality or the single-assignment/backtracking access mode of logical variables. Therefore, in most cases each variable appears exactly once in a given pattern, complex disjunctions aside.

After a successful match, variables may be bound to subobjects of the matched target datum. Whether a certain variable is bound or not may depend on the chosen alternative of a disjunction. The user is fully responsible for reading only bound variables.

Since variables have no distinguished initial "unbound" state, there is no *dynamic* check whether a variable has been bound by the most recent matching attempt: unsuccessful matches leave the occurring variables in unspecified state. Fortunately, the *static* effect of a pattern on given variables can be inferred inductively from its logical structure; the inference rules are available both as user documentation and runtime queries; details are beyond the scope of this article.

The advantage of this form of variable binding is that initialization costs are minimal and patterns can be reused without special preparation. It also implies that it is transparent to the user which branch of a disjunction has been taken; observed values of variables cannot be used to reconstruct the information. While this is the desired abstraction in most cases, special "marked" forms of disjunction (implemented by classes `IntBranch` and `EnumBranch`) can be used to retain the information for complex nested searches.

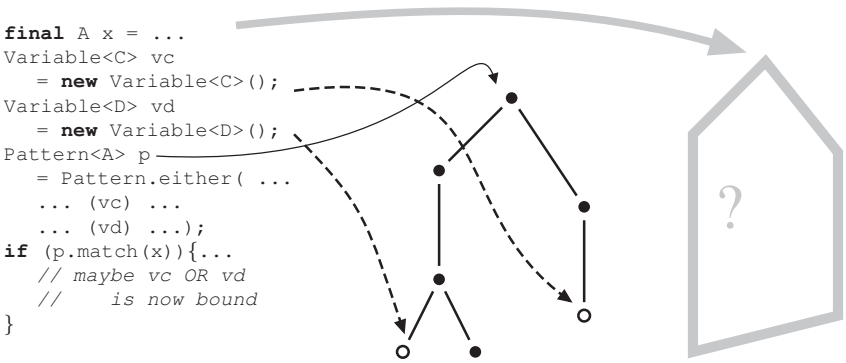


Figure 2: Explicit references to data, variables and pattern are required.

The restricted role of variables, although rather poor from a high-level declarative viewpoint, mimics closely the behaviour of local variables and fields in object-oriented programming, and should therefore feel natural to the programmer.

3.7 Encapsulated Search

For patterns with a single variable, bindings for all matches can be collected eagerly or lazily with `eagerBindings(Pattern)` and `lazyBindings(Pattern)`, respectively, thus effecting fully reified encapsulated search as strongly typed objects of the Java collection framework. The iteration pattern for all matches of pattern `p` for target `x` simplifies accordingly, with operational semantics equivalent to the loop given in section 3.4:

```
for (C c : vc.lazyBindings(p, x))
  doSomething(c) ;
```

The alternative is to calculate all possible matches before any processing, i.e. eagerly. This works of course only if the number of possible matches is finite:

```
List<A> list = vc.eagerBindings(p, x);
```

In the current implementation, simultaneous encapsulated search for multiple variables is not implemented, because of the lack of support for ad-hoc tuple types in Java.

3.8 Backtracking and Reentrance

The paradigmatic platform for backtracking nondeterminism is of course the Prolog language. Implementation hints can be gleaned from its operational semantics, in particular Warren's Abstract Machine (WAM) [War83].

The WAM uses to less than four categories of stack space (of which call stack and choice stack are interleaved to form the so-called local stack) to control nondeterminism and vari-

able bindings. Fortunately for us, the simpler nature of imperative variables in Paisley and the underlying Java Virtual Machine (JVM), where no variable bindings, let alone object allocations, need to be undone by backtracking, allows a significant reduction of complexity: Besides the regular call stack, only a choice stack for yet unexplored disjunctive branches is required. And because the latter is strictly local to each matching, it can be implemented decentrally and transparently, distributed and hidden in the pattern combinator instances themselves. In contrast to an interleaving implementation, this requires no privileged access to the JVM stack, and is hence easily portable to other platforms.

The disjunctive pattern `either(p, q)` behaves as `p` initially, but switches to `q` after solutions to `p` are exhausted. The conjunctive pattern `both(p, q)` fixes a solution for `p` and produces solutions for `q`, but retries `p` and resets `q` whenever solutions to the latter are exhausted. These modes of operation subsume plain Boolean combination for deterministic argument patterns, and result in the concatenation and Cartesian product (in lexical order), respectively, of solutions for independent argument patterns. Overall behaviour can be more complex if `q` depends on `p` via observation of variable bindings or side effects; see section 3.12 below.

Note that a naive implementation of conjunction in terms of sequential matching attempts would not result in a full implementation of backtracking, but rather in the partial backtracking found in many simple search algorithms: The first argument pattern is committed to a solution before the second is tried; the latter is not reiterated for alternative solutions of the former. Contrast with the **`else`** branch in the code of Figure 3 below.

To illustrate the operation of the backtracking mechanism in detail, Figure 3 shows parts of the implementation of the `both` combinator. The local choice stack segment records whether the `left` argument pattern has succeeded and a reference to the matched target. The `match` method attempts to pair a solution for each argument pattern, and sets up the choice stack for later retrieval by `matchAgain()` as a side effect. The implementation of `matchAgain()` is almost identical, except that the grey statements are omitted and the underlined function head and recursive call are replaced by `matchAgain()`. The `either` combinator implements disjunction in an analogous manner.

As a consequence of the residual implementation of the choice stack, patterns are not thread safe: They can be reused sequentially, as required for `both` (see Figure 3), but not concurrently. On the upside, this allows for a simple, local and therefore highly efficient implementation of the cut: Patterns implement a method `cut()` that discards unused solutions with minimal effort, and a method `clear()` that additionally purges obsolete references from the choice stack, in order to control heap space usage even if pattern objects remain live for long times. Both methods descend recursively to subpatterns.

3.9 Specialized Pattern Libraries

The current implementation of Paisley comes with a few more application-specific groups of pattern combinators. In particular, the static factory class `XMLPatterns` supports content extraction and navigation along all axes of a W3C XML document object model

```
private Pattern left, right;                    // pattern tree
private A       target_save;                    // local choice stack
private boolean left_matched;                   // local choice stack

public boolean match(A target) {
  if (left_matched = left.match(target)) {
    target_save = target;                      // for use by matchAgain()
    if (right.match(target)) return true;
    else
      while (left_matched = left.matchAgain())
        if (right.match(target_save)) return true;
    target_save = null;                        // solutions exhausted
  }
  return false;
}
```

Figure 3: Implementation of backtracking (excerpt)

(DOM [HHW+00]). It shows compositional abstraction through pattern lifting, and non-invasive "patternification" of an existing, abstract data model, namely the standard Java package `org.w3c.dom`. Showcase examples using this and other pattern factories are included in the Paisley download package at [TL11]. Figure 4 shows an excerpt that deals with "glossary entries" in an XHTML document, that is adjacent *term–description* (`<dt>`–`<dd>`) pairs in a *definition list* (`<dl>`). In particular, the self-contained example code extracts from a document the full relation between terms and *hyperlinks*, that is *anchors* (`<a>`) with a `href` attribute, contained in the respective following descriptions. Note how the implementation is, mandatory formal noise of Java aside, hardly more complicated than the prose description.

This example shows the classical way of using patterns, where the variables appear in leaf position, and the context is specified ("generate"). Paisley supports the symmetric situation where the content of the bound structures is narrowed further ("test"). For instance, extending the pattern `r` to

```
glossaryPair(both(dt, textContent(startsWith("c"))),
             descendant(anchorWithHRef
                          (both(href, not(startsWith("#"))))  ))
```

will match only glossary entries starting with lower case "c" and anchors to non-local hrefs.

For other, user-defined tasks the implementation strategy is similar: encapsulating all dirty details of testing, iterating, backtracking and cutting into library patterns, thus creating a clean basis on which the operational code can be formulated in an intentional, declarative way.

```
import eu.bandm.tools.paisley.*;
import static eu.bandm.tools.paisley.Pattern.*;
import static eu.bandm.tools.paisley.XMLPatterns.*;
import org.w3c.dom.*;

class XML_example {
  final static String XHTMLNS = "http://www.w3.org/1999/xhtml";
  static Pattern<Node> xhtmlElement(String localName,
                                    Pattern<? super Element> element) {
    return element(both(name(XHTMLNS, localName), element));
  }
  static Pattern<Node> glossaryPair(Pattern<? super Element> dt,
                                    Pattern<? super Element> dd) {
    return both(xhtmlElement("dt", dt),
                nextSibling(xhtmlElement("dd", dd)));
  }
  static Pattern<Node> anchorWithHRef(Pattern<? super String> href) {
    return xhtmlElement("a", attrValue("href", href));
  }
  final Variable<Element> dt = new Variable<Element>();
  final Variable<String> href = new Variable<String>();
  final Pattern<Element> r =
    glossaryPair(dt, descendant(anchorWithHRef(href)));
  final Pattern<Document> p = root(descendantOrSelf(r));

  { // ... let "doc" be a w3c dom representation of an XHTML document
    if (p.match(doc)) do {
      System.out.println("Glossary␣entry␣\""
                         + dt.getValue().getTextContent()
                         + "\"␣refers␣to␣\"" + href.getValue() + "\"");
    } while (p.matchAgain());
  }
}
```

Figure 4: XHTML Glossary Entry Example

3.10 Projection and Testing

The base case of data extraction is a conceptual total function $f : A \to B$; the archetypal example being a getter method in class `A` which reads some member field of type `B`. This induces a function that maps each pattern `p` of type `Pattern<B>` to a pattern of type `Pattern<A>` that matches a target x by having `p` match $f(x)$. Given a suitable reification `f` of f, this can be written in Paisley simply as `transform(f, p)`.

The more general case is that of partial functions, where an undefined result causes the matching to fail. These are realized conveniently as subclasses `T` of `Transform<A, B>`, which implement the **`boolean`** `isApplicable(A)` and `B apply(A)` methods explicitly, with the obvious semantics.

In principle, `Transform` is a complete basis for projection and testing, that is for all functional pattern components except variables. Of course, hand-coded projections/tests can be defined by the user, where more convenient or efficient for the application at hand.

3.11 Pattern Substitution and Closure

Apart from extracting data from a match, variables also play a meta-level role as hooks for pure pattern algebra, thus enabling powerful generic abstractions and transformations.

The most basic one is substitution, which enables pattern parametrization: In the code fragment

```
Variable<V> v = new Variable<V>();
Pattern<R> top = ... (v) ... ;
Pattern<V> sub = ... ;
Pattern<R> newTop =  v.bind(top, sub);
```

`newTop` denotes a pattern in which every occurrence of `v` is replaced by a reference to `sub`. The implementation works non-invasively by recursively nested matching; hence, even hand-coded patterns which hide their logical structure (if any) can safely be substituted into. Duplication of substitution replacements is avoided by exploiting sequential reusability of patterns. As a moniker for the programming interface, read `v.bind(p, q)` as the lambda calculus expression $(\lambda v.p)q$.

In extension of the quantifier perspective on patterns, variables also serve as the bridgehead of "star" or "plus" Kleene closures:

```
Variable<V> v = new Variable<V>();
Pattern<V> once = ... (v) ... ;
Pattern<V> some = v.star(once);
Pattern<V> more = v.plus(once);
```

The newly constructed patterns `some`/`more` are the star/plus closures, respectively, of the path relation between `once` and `v`, insofar as they obey the expected mutually recursive behavioral equivalence relation

`some` $\equiv$ `either(v, more)` `more` $\equiv$ `v.bind(once, some)`

Using these, the complex pattern constructor `XMLPatterns.descendantOrSelf(p)` featured in Figure 4 can be defined concisely as

```
v.bind(v.star(child(v)), p)
```

where `v` is a fresh variable, and the primitive `child` is implemented in terms of the canonical `org.w3c.dom.Node` getters `getFirstChild()` and `getNextSibling()`. Note how the two sources of nondeterminism, regarding horizontal (`child`) and vertical (`star`) position in the document tree, respectively, combine completely transparently.

3.12 Breaking the Paradigm

The "pragmatic philosophy" of Paisley is to leverage a declarative style of writing and thinking, while the code "behind the scenes" remains genuinely imperative, with no intermediate transformation.

Whether any knowledge of the implied control flow is considered part of pattern semantics, is a fundamental decision on the level of "coding style guidelines", and its consequences must be considered carefully. Here clearly a Rubicon would be crossed.

On the upside, by taking advantage of sequential execution order, powerful functionalities like non-linear patterns can be implemented. The following pattern `p`, built using the Paisley XML facilities discussed in section 3.9, matches all XHTML anchors which contain their target URL literally in their text content:

```
Variable<String> v = new Variable<String>();
Pattern<Element> p =
  all(name(XHTMLNS, "a"),
      attrValue("href", v),
      descendant(textContent(contains(v.getValue()))));
```

On the downside, it is evidently easy to accidentally break the declarative matching semantics, resulting in all the kinds of subtle and hard-to-debug program behaviour we set out to avoid with our design in the first place. Extending the library in this way, while technically supported and perhaps pragmatically valid, is certainly a different use case altogether; the two should be distinguished carefully for fundamental software engineering reasons.

4 Conclusion

4.1 Related Work

A theoretically elegant design of pattern matching capabilities for Java, JMatch, is presented in [LM03]. While it has had much impact, and is cited heavily by later work, there are severe drawbacks: The approach assumes a perspective on pattern matching that is very much like logical programming. As a result, their nondeterminism is rather heavy-weight: It requires CPS transformation of certain program parts, and hence interferes severely with apparent control flow, making program understanding and debugging forbiddingly difficult. Furthermore, the solution is a host language extension and requires a special academic compiler. All such experiments are eventually doomed to oblivion unless some big vendor adopts the technology.

As mentioned above, the multi-paradigm language Scala [OSV10] incorporates a powerful pattern matching idiom with clean semantics and user-defined extensibility, via singleton objects and the `unapply` method [EOW07]. Being part of the core Scala design, it is better integrated with the host language than our approach can ever hope to be. On the other hand, we find the lack of nondeterminism and pattern algebra significant weaknesses.

4.2 Comparative Evaluation Framework

Concerning the evaluation of the design of different pattern matching mechanisms, a paper [EOW07] from the Scala context introduced a grid of nine criteria. Omitting the last three, which deal with concrete performance measuring and cannot easily be reconstructed, we find that Paisley corresponds largely to the "extractor" type of solution discussed in that paper, with some notable differences.

Conciseness of the Framework Programming overhead is required for the Paisley projection operations, see section 3.10 above. They correspond to the "extractors" in Scala, and are a little more verbose than those, obviously due to the less expressive host language syntax. Furthermore, the chain of delegation to embedded Paisley patterns must always be written down explicitly, whereas in many situations a call to `unapply` will silently be inserted in the Scala approach.

The disadvantages of Paisley are offset to some degree by the superior abstraction capabilities, whereby for many applications predefined, highly generic library building block for patterns are provided ready-to-use. Additionally, the Paisley approach does not share the weakness of Scala pattern matching that descent into the structure of an object is reified, and hence boxed and unboxed, at every level of `unapply`.

Conciseness of Shallow/Deep Matches The syntax of a concrete application of a complex Paisley pattern has least possible syntactic noise. A source of noise that cannot be evaded by the nature of our design is the absence of a *rule* concept in Paisley: There is no equivalent of the Scala matching block that follows a `match` operator. Instead, the scheduling of alternative patterns (rule left-hand sides) and the association of corresponding reactions (rule right-hand sides) is expressed in the hosting object-oriented style.

Maintainability: Representation Independence No internal representation at all need be revealed, because only the functional testing/projection interfaces have to be implemented. See the XML DOM example cited in Section 3.9 above. Nevertheless, in many cases data abstraction is trivial, so the extractors will follow the internal structure naturally.

Maintainability: Extending (Data) Variants / Patterns The data and the pattern world may grow arbitrarily without mutually affecting the behaviour of older model class definitions and patterns. Since patterns are only defined by their terse functional interface (see Section 3.4), arbitrary new variants can be added, and existing combinations can freely be abstracted at any time.

In terms of compositionality, this goes far beyond Scala extractors: Being transparent regarding nondeterminism, also arbitrary disjunctions and even encapsulated searches can be abstracted to self-contained pattern components.

References

[BC08] Becchi, M.; Crowley, P.: Extending finite automata to efficiently match Perl-compatible regular expressions. In Proc. 2008 ACM CoNEXT Conference (CoNEXT '08). ACM, New York, 2008; S. 25:1–25:12.

[BGJ11] Blomer, J.; Geiß, R.; Jakumeit, E.: The GrGen.NET User Manual. http://www.grgen.net, 2011.

[CD99] Clark, J.; DeRose, S.: XML Path Language (XPath) Version 1.0. W3C, http://www.w3.org/TR/1999/REC-xpath-19991116/, 1999.

[EB10] Ebert, J.; Bildhauer, D.: Reverse Engineering Using Graph Queries. In (Schürr, A.; Lewerentz, C.; Engels, G.; Schäfer, W.; Westfechtel, B. Hrsg.) Graph Transformations and Model Driven Engineering, LNCS 5765. Springer Verlag, 2010.

[EOW07] Emir, B.; Odersky, M.; Williams, J.: Matching Objects with Patterns. In (Ernst, E. Hrsg.) Proc. 21st ECOOP, LNCS 4609. Springer, 2007.

[GL06] Genèves, P.; Layaïda, N.: A System for the Static Analysis of XPath. ACM Trans. Inf. Sys. 24, 2006.

[HHW+00] Le Hors, A.; Le Hégaret, P.; Wood, L.; Nicol, G.; Robie, J.; Champion, M.; Byrne, S.: Document Object Model (DOM) Level 2 Core Specification Version 1.0. W3C, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/, 2000. Recommendation.

[HP00] Hosoya, H.; Pierce, B.C.: XDuce: A Typed XML Processing Language. In Proc. 3rd Workshop on the Web and Data Bases (WebDB 2000), S. 226–244. Springer Verlag, 2000.

[LM03] Liu, J; Myers, A.C.: JMatch: Iterable Abstract Pattern Matching for Java. In Practical Aspects of Declarative Languages, LNCS 2562. Springer, 2003.

[LT11] Lepper, M.; Trancón y Widemann, B.: Optimization of Visitor Performance by Reflection-Based Analysis. In (Cabot, J.; Visser, E. Hrsg.) Theory and Practice of Model Transformations, LNCS 6707. Springer Verlag, 2011.

[OSV10] Odersky, M.; Spoon, L.; Venners, B.: Programming in Scala. artima, 2nd edition, 2010.

[TL11] Baltasar Trancón y Widemann and Markus Lepper. *Paisley Download and Documentation Page*. http://www.bandm.eu/metatools/docs/usage/paisley-download.html, 2011.

[TL12] Baltasar Trancón y Widemann and Markus Lepper. Paisley: pattern matching à la carte. In Proc. 5th International Conference on Model Transformation (ICMT 2012), LNCS 7307. Springer Verlag, 2012; S. 240–247.

[War83] Warren, D.: An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, 1983.

DFF – Design For Future 2013
Software Engineering für langlebige Systeme

Wiederverwendbarkeit von Migrationswissen durch Techniken der modellgetriebenen Softwareentwicklung

Marvin Grieger, Stefan Sauer

Universität Paderborn
s-lab – Software Quality Lab
Zukunftsmeile 1
33102 Paderborn
mgrieger@s-lab.upb.de
sauer@s-lab.upb.de

Abstract: Softwaresysteme müssen auf Grund sich ändernder Anforderungen fortwährend gepflegt und weiterentwickelt werden. Eine Migration in eine neue Umgebung, die auf modernen Softwaretechnologien basiert, ist oftmals der einzige Ausweg, die neuen Anforderungen vollständig zu erfüllen. Viele Unternehmen versuchen dennoch, eine Migration zu vermeiden, da sie mit einem hohen wirtschaftlichen Investment und Risiko verbunden ist. Dies ist insbesondere bedingt durch die fehlende Werkzeugunterstützung in der Durchführung der Migration. In diesem Beitrag beschreiben wir diesbezüglich ein konkretes Fallbeispiel aus der Praxis. Darauf aufbauend entwickeln wir einen Migrationsprozess, der durch die Nutzung von Semi-Automatismen basierend auf der Formalisierung und Wiederverwendung des in der Migration entstehenden Wissens in Form von Modellen und Transformationen gekennzeichnet ist.

1 Einleitung und Motivation

Softwaresysteme altern über die Zeit. Dieser Alterungsprozess ist dadurch gekennzeichnet, dass sich die Kluft zwischen den Anforderungen an die Systeme und deren tatsächliche Leistungsfähigkeit immer weiter vergrößert [SWH10].Oftmals erfolgt eine Erhaltung und Weiterentwicklung der Systeme, um dem Alterungsprozess entgegenzuwirken. Über die Zeit geht dies mit einer verringerten Qualität der Software und einer steigenden Wartungskomplexität einher. Der Umfang der Änderungsmöglichkeiten durch eine Weiterentwicklung ist jedoch begrenzt, gleichzeitig können Einschränkungen der zu Grunde liegenden Technologie dazu führen, dass es nicht möglich ist, alle Anforderungen umzusetzen. Ein pragmatischer Ausweg ist laut Sneed die Migration des Systems in eine neue Umgebung [SWH10].
Die Motivation, eine Migration anstelle einer Neuentwicklung durchzuführen, ist die Erhaltung des in die Software eingeflossenen Wissens und so auch der Schutz der in das System getätigten Investitionen. Eine manuelle Überführung dieses Wissens birgt jedoch erhebliche Risiken bedingt durch die Größe und Komplexität der Softwaresysteme. Durch die Entwicklung von Werkzeugen und Techniken, die diesen Schritt unterstützen,

können Kosten gespart und Fehler minimiert werden. Die Entwicklung solcher Werkzeuge und Techniken untersuchen wir innerhalb eines Forschungsprojekts.
In diesem Beitrag gehen wir zunächst im Detail auf ein Fallbeispiel aus der Praxis ein. Darin beschreiben wir die Herausforderungen in der Migration und leiten Anforderungen an eine Unterstützung ab. Das Fallbeispiel nutzen wir im Rahmen des Forschungsprojekts als Ausgangspunkt für die Entwicklung des Migrationsprozesses und zur Evaluierung unseres Ansatzes. Anschließend stellen wir den von uns verfolgten Ansatz vor und beschreiben verwandte Arbeiten.

2 Fallbeispiel

Im Rahmen eines Forschungsprojekts kooperieren wir mit einem Industriepartner, der sich auf die Entwicklung und den Vertrieb von datenbankbasierten Anwendungen in der Logistik-Domäne spezialisiert hat. Dabei untersuchen wir die Migration von Applikationen, die für die Plattform Oracle Forms 6i entwickelt wurden. Lange Zeit herrschte Unsicherheit darüber, inwiefern die Plattform auch in Zukunft weiterentwickelt werden würde, da der Hersteller verstärkt die Entwicklung alternativer Produkte in den Vordergrund stellte. Anfang des Jahres wurde diesbezüglich eine Stellungnahme der Firma Oracle veröffentlicht, in der die langfristigen Strategien für die vorhandenen Plattformen offengelegt wurden [Or12]. In dieser wird bestätigt, dass auch in Zukunft eine Weiterentwicklung der Forms-Plattform auf Grund der großen Installationsbasis stattfinden wird. Gleichzeitig werden Empfehlungen abgegeben, welche Plattformen sich als potenzielle Zielplattform für eine Migration eignen. Dies umfasst unter anderem die Plattform Oracle ADF, welche wir als Zielplattform betrachten. Dabei wird auch klargestellt, dass seitens der Firma Oracle keine Werkzeugunterstützung für eine solche Migration bereitgestellt wird. Begründet wird dies mit den Unterschieden zwischen Quell- und Zielplattform. Die wichtigsten Unterschiede haben wir nachfolgend zusammengefasst:

1. *Programmiersprache*
   Die Programmiersprache der Quellplattform Oracle Forms ist PL/SQL, welche proprietär und prozedural aufgebaut ist. Die Programmiersprache der Zielplattform Oracle ADF hingegen ist die objektorientierte Programmiersprache Java.

2. *Architektur*
   Während Applikationen der Quellplattform monolithisch aufgebaut sind, werden Anwendungen in der Zielplattform nach dem Model-View-Controller-Architekturmuster strukturiert. Folglich muss eine Dekomposition der Quellapplikation durchgeführt werden.

3. *Paradigma*
   Dem Design der Quell- und Zielplattform liegen unterschiedliche Prinzipien zu Grunde. Beispielsweise wird in der Zielplattform eine Prozessorientierung verfolgt, während Applikationen der Quellplattform datenorientiert entwickelt werden. Ein weiteres Beispiel ist die Funktionalität, die durch eine Datenbank als Komponente innerhalb der Applikation bereitgestellt wird. Während die Datenbank in der Zielplattform lediglich als Datenspeicher genutzt werden soll, stellt sie in der Quellap-

plikation eine zentrale serverseitige Komponente dar, auf der beispielsweise Validierungen durch Prozeduren (sog. Stored Procedures) ausgeführt werden. Solche Paradigmenwechsel erschweren die Überführung auf der technischen Ebene.

Der Bedarf, bestehende Forms-Applikation nach Oracle ADF zu migrieren ist dennoch vorhanden. Dies wird besonders durch die Präsenz des Themas Migration innerhalb der Oracle Community deutlich[1]. Die Gründe für eine Migration sind dabei vielseitig. Viele Unternehmen sind weiterhin bzgl. der langfristigen Entwicklung der Forms-Plattform verunsichert. Zudem besteht der Wunsch, leistungsfähige Entwicklungsumgebungen sowie aktuelle Softwaretechnologien und -architekturen einzusetzen. Dadurch soll die Wartbarkeit der Systeme erhöht und eine schnellere Anpassungsfähigkeit an neue Anforderungen ermöglicht werden.
In diesem Kontext untersuchen wir als Fallbeispiel die Migration einer größeren Forms-Applikation. Sie wurde über mehrere Jahre entwickelt und umfasst ca. vier Millionen Zeilen Code. Um diese umfangreiche Migration zu unterstützen, erarbeiten wir im Rahmen unseres Forschungsprojekts einen werkzeuggestützten Migrationsprozess basierend auf Techniken der modellgetriebenen Softwareentwicklung. Dazu definieren wir zunächst die Anforderungen an diesen Prozess, in den unsere Erfahrungen mit auf dem Markt befindlichen Lösungen eingeflossen sind:

a) *Wiederverwendbarkeit*
Über unseren Industriepartner erhalten wir Zugriff auf mehrere Applikationen der Quellplattform, die von unterschiedlichen Entwicklergruppen entwickelt wurden. Das Wissen, welches während der Migration einer Applikation aufgebaut wird, soll ebenfalls in nachfolgenden Migrationsprojekten genutzt werden können. Dazu muss dieses Wissen explizit erfasst werden, beispielsweise in Form eines Katalogs von Templates bzw. Regeln, der dann kundenspezifisch genutzt, angepasst und erweitert werden kann. Dies ist ein zentraler Aspekt unseres Migrationsprozesses.

b) *Automatismen*
Die Nutzung von Automatismen in der Migration bietet erhebliche Vorteile. So können insbesondere für wiederkehrende Tätigkeiten Kosten gespart und gleichzeitig manuelle Implementierungsfehler ausgeschlossen werden. Dabei müssen jedoch auch die Grenzen der Automatisierbarkeit bekannt sein. Oftmals erzwingen Werkzeuge einen hohen Grad an Automatisierung, der die Qualität des Resultats verringert, da applikationsspezifische Eigenheiten nicht berücksichtigt werden. Dies ist insbesondere im Rahmen einer 4GL-Migration der Fall, d.h. einer Migration ausgehend von einer Umgebung einer Vierten Generationssprache (4GL). Werkzeuge stützen sich in diesem Fall auf die proprietären, vorgegebenen Strukturen einer solchen Applikation und ignorieren applikationsspezifische Erweiterungen. In unserem Migrationsprozess möchten wir aus diesen Gründen eine semi-automatisierte Werkzeugunterstützung bereitstellen, in der insbesondere Flexibilitätspunkte für applikationsspezifische Anpassungen bereitstehen.

[1] http://www.doag.org/

c) *Abstraktion*
Eine Transformation der Applikation auf Basis der reinen syntaktischen Informationen ist an vielen Stellen nicht zielführend. Durch Einschränkungen der Quellplattform mussten zum einen Funktionalitäten der Applikation explizit ausprogrammiert werden, wobei die Freiheitsgrade der Quellplattform ausgenutzt und vorgegebene Strukturen zweckentfremdet wurden. Zum anderen sollen in der Migration die Möglichkeiten der Zielplattform ausgenutzt werden, zum Beispiel in der Überführung der Benutzungsoberfläche: In der Quellplattform sind die Elemente auf der Oberflächen absolut positioniert, während in der Zielplattform eine relative Positionierung angewendet wird. Zusätzlich existieren in der Zielplattform neue Visualisierungsmöglichkeiten der Datensätze, z.B. in Form von Diagrammen. Auf Grund dieser Unterschiede wird eine direkte 1:1-Überführung nicht verfolgt, vielmehr sollen die relevanten Informationen einer Benutzungsoberfläche extrahiert, in einer Zwischenrepräsentation abgebildet und in der Zielplattform umgesetzt werden.

d) *Individuelle Zielarchitektur*
Im Rahmen der Migration muss die Architektur für die Applikation in der Zielplattform (Zielarchitektur) entworfen werden. Diese unterliegt einem Spannungsfeld zwischen der optimalen Zielarchitektur und der vorhandenen Altanwendung [ZGW10], weshalb die Erarbeitung der Zielarchitektur ein manueller Prozess ist, der ein tiefes Verständnis sowohl der Altanwendung (einschließlich der Quellplattform) als auch der Zielplattform voraussetzt. Sobald ein Werkzeug eine vollautomatische Migration anstrebt, wird die Zielarchitektur meistens komplett vorgegeben. Dies ist darin begründet, dass Umsetzungsentscheidungen Seiteneffekte haben können, die andere Umsetzungsmöglichkeiten beeinflussen oder ausschließen. Durch unseren Migrationsprozess möchten wir die Definition einer individuellen Zielarchitektur ermöglichen. Gleichzeitig sollen bewährte Architekturen bzw. Umsetzungsmöglichkeiten wiederverwendet werden können.

e) *Prozessorientierung*
Applikationen der Zielplattform werden ausgehend von den zu Grunde liegenden Geschäftsprozessen entworfen, wodurch auf technischer Ebene eine Strukturierung in wiederverwendbare Arbeitsabläufe und -schritte erfolgt. Das Konzept eines prozessorientierten Ansatzes ist in der Quellplattform nicht vorhanden, die konkreten Abläufe, die auf Grund eines Geschäftsprozesses durchgeführt werden, sind auf technischer Ebene nicht sichtbar. In der Migration muss die Prozessorientierung berücksichtigt werden, um die Zielapplikation entsprechend den Konzepten der Zielplattform sinnvoll zu strukturieren. Dies kann beinhalten, Arbeitsabläufe explizit zu erfassen, um zusätzliche Strukturierungsinformationen bereitzustellen.

Diese Anforderungen werden durch bestehende Werkzeuge und die zugehörigen Migrationsprozesse nicht erfüllt. Deshalb entscheiden sich viele Unternehmen, die den Bedarf haben, ihre Altanwendung zu modernisieren, für eine Weiterentwicklung. Im Rahmen des Forschungsprojekts untersuchen wir, inwiefern Techniken der modellgetriebenen Softwareentwicklung dazu genutzt werden können, die benötigte Flexibilität zur Entwicklung eines individuellen Migrationspfades und gleichzeitig die Werkzeugunterstützung in der Überführung von Altanwendungen bereitzustellen.

3 Ansatz

Techniken der modellgetriebenen Softwareentwicklung wurden bereits in verschiedenen Projekten im Kontext der Softwaremigration eingesetzt und untersucht (vgl. Abschnitt 4). Die meisten dieser Arbeiten adressieren jedoch nur die einmalige Migration eines spezifischen Softwaresystems. Im Rahmen unserer Forschung möchten wir ein Verfahren entwickeln, welches das während der Migration entstandene Wissen explizit festhält. Dies umfasst sowohl das Wissen über den Aufbau der resultierenden Applikation, welches durch Modelle auf verschiedenen Abstraktionsebenen repräsentiert wird, als auch das Wissen über die Zusammenhänge in der Migration, welches für spätere Migrationen von Applikationen derselben Plattform in Form einer Werkzeugunterstützung genutzt werden kann. In einer früheren Arbeit haben wir unseren Ansatz in Form einer artefaktbasierten Sicht auf den Migrationsprozess beschrieben [GGS12]. In diesem Beitrag möchten wir auf Teilaspekte dieses Prozesses im Detail eingehen.

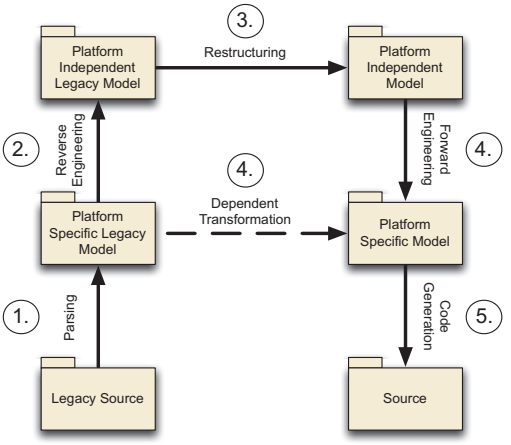


Abbildung 1: Modelle und Transformationen im Migrationsprozess

In Abbildung 1 ist ein Ausschnitt unseres Migrationsprozesses dargestellt, angelehnt an das Reengineering-Hufeisen [KWC98]. Die grundlegende Idee des Ansatzes ist die Nutzung von Modellen unterschiedlicher Abstraktionsebenen, die durch Modelltransformationen ineinander überführt werden. Die Abstraktionsebenen orientieren sich dabei an dem MDA-Ansatz der Object Management Group (OMG). Dabei werden alle Modelle in einem zentralen Repository gespeichert und über Traceability-Informationen der Transformationen miteinander verknüpft. Nachfolgend wird auf die Rolle der verschiedenen Modelle und der Transformationen separat eingegangen:

1. *Modellbildung*
   Zunächst wird die Applikation in Form eines plattformspezifischen Modells (*Platform Specific Legacy Model*) erfasst. Dies umfasst eine verlustfreie Abbildung der Quellartefakte, welche strukturell in kleinste Einheiten zerlegt werden, indem zugehörige Parser die Syntaxbäume der enthaltenen Code-Blöcke berechnen. Schnittstel-

len externer Komponenten, mit denen die Applikation kommuniziert, werden in diesem Schritt ebenfalls erfasst. Zusätzlich können Ergebnisse automatisch durchgeführter dynamischer Analysen einfließen, um beispielsweise Aufrufreihenfolgen oder Ausführungszeiten zu ermitteln. Nachdem die initiale Abbildung der Applikation erfolgt ist, müssen implizit vorhandene Beziehungen zwischen Modellelementen explizit sichtbar gemacht werden. Im Rahmen des Fallbeispiels konnten wir feststellen, dass viele Beziehungen auf Quellcode-Ebene nicht direkt sichtbar waren, sondern erst durch das Wissen über das Laufzeitverhalten der Quellplattform deutlich wurden. Das Modell wird um diese Informationen angereichert. Auf Grund des erhöhten Informationsgehalts des Modells gegenüber den Quellartefakten werden Letztere nicht mehr als Grundlage für weitere Migrationsschritte verwendet.
Die Modellbildung läuft für unterschiedliche Applikationen derselben Plattform identisch ab, so dass eine Routine entwickelt werden kann, die diesen Schritt vollautomatisch durchführt.

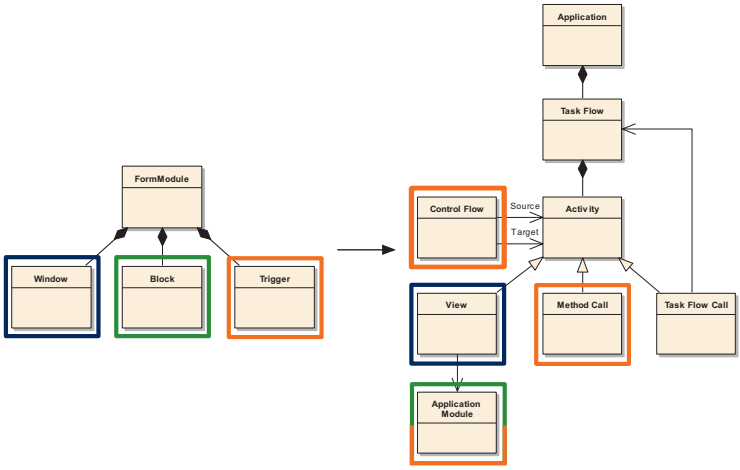


Abbildung 2: Ausschnitte des plattformspezifischen Metamodells der Quell- (links) und Zielplattform (rechts) sowie die Abbildung zwischen einzelnen Elementen (Farbe)

2. *Abstraktion*
Nachfolgend wird ein Abstraktionsschritt durchgeführt, in welchem eine Zwischenrepräsentation der Quellapplikation erzeugt wird. Um die Notwendigkeit für diesen Schritt zu verdeutlichen, wird an dieser Stelle ein konkretes Beispiel angeführt.
In Abbildung 2 ist auf der linken Seite ein Ausschnitt des plattformspezifischen Metamodells der Quellplattform dargestellt. Durch den Modellbildungsschritt wurde die Applikation konform zu diesem Metamodell erfasst. Die Klasse *FormModule* repräsentiert ein abgeschlossenes Modul der Oracle Forms Plattform, alle Strukturen innerhalb eines Quellartefakts sind dieser Klasse untergeordnet. Eine Anwendung besteht aus einer Menge von Quellartefakten. Die *Window*-Klasse beschreibt die enthaltenen Oberflächen, während die *Block*-Klasse die angebundene Datenhaltung

abbildet. Des Weiteren existiert eine Menge an *Trigger*-Klassen, welche Code-Blöcke in der Programmiersprache PL/SQL enthalten.
Auf der rechten Seite ist das plattformspezifische Metamodell der Zielplattform dargestellt. Eine Anwendung ist in mehrere Applikationen unterteilt, dargestellt durch die Klasse *Application*. Eine Applikation enthält einen oder mehrere Arbeitsabläufe in Form von *Task Flows*, die sich aus einer Menge an Aktivitäten zusammensetzen. Eine Aktivität umfasst z.B. die Anzeige einer Benutzungsoberfläche (*View*), in der die angebundenen Datensätze (*Application Module*) angezeigt werden. Weitere Aktivitäten repräsentieren Methodenaufrufe (*Method Call)* oder die Ausführung eines Arbeitsablaufes (*Task Flow Call*). Der Kontrollfluss zwischen einzelnen Aktivitäten wird explizit dargestellt (*Control Flow*).
Im Kontext der Migration werden Instanzen des Metamodells der Quellplattform in Instanzen des Metamodells der Zielplattform überführt, so dass die Funktionalität der Anwendung erhalten bleibt. Dies wird durch die Definition von Modelltransformationen in Form von Abbildungen zwischen den Metamodellen realisiert. Eine Voraussetzung für die Definition einer solchen Abbildung ist, dass beide Metamodelle ein Konzept in einer ähnlichen Granularität beschreiben. Dies ist in dem Beispiel sowohl bzgl. der Benutzungsoberflächen als auch der Datenanbindung der Fall, beide Konzepte werden sowohl in der Quell- als auch in der Zielplattform explizit repräsentiert. Dadurch ergibt sich eine Abbildung von *Window* auf *View* (blau) sowie von *Block* auf *Application Module* (grün).
Der Wechsel von einer Benutzungsoberfläche auf eine andere wird in der Zielplattform durch einen zugehörigen Kontrollfluss ebenfalls explizit beschrieben. In der Quellplattform hingegen wird diese Navigation programmatisch durch einen Code-Block innerhalb eines *Triggers* durchgeführt. Gleiches gilt für andere Konzepte, wodurch Teile der Code-Blöcke auf unterschiedliche Klassen abgebildet werden müssen (orange). Die syntaktische Zerlegung eines Code-Blocks resultiert jedoch lediglich in einer Menge an Anweisungen, welche nicht näher klassifiziert sind. Folglich muss eine semantische Interpretation durchgeführt werden, um die relevanten Informationen zu extrahieren und in der benötigten Granularität bezüglich der Zielplattform zu erfassen.
In unserem Ansatz wird solch eine Zwischenrepräsentation der Applikation explizit erstellt (*Platform Independent Legacy Model*). Dazu wird eine Beschreibungssprache in Form eines Metamodells bereitgestellt, durch welche unterschiedliche Applikationen in einer einheitlichen Sprache repräsentiert werden können. Dadurch können Analyse- oder Transformationsverfahren, welche im Rahmen der Migration einer spezifischen Applikation entwickelt wurden, auch für die Migration anderer Applikationen wiederverwendet werden. Das Metamodell der Sprachdefinition ermöglicht die Beschreibung verschiedener Aspekte der Applikation auf unterschiedlichen Abstraktionsebenen; ein plattformunabhängiges Modell untergliedert sich intern folglich entsprechend dieser Ebenen. Dies umfasst unter anderem den strukturellen Aufbau der Benutzungsoberflächen und die Möglichkeit, architektonische Sichten der Applikation zu erstellen. Dabei werden explizit plattform- und domänenspezifische Aspekte berücksichtigt. Gleichzeitig kann das Metamodell um applikationsspezifische Besonderheiten in einer Migration situativ oder kundenspezifisch erweitert werden. Ein Beispiel ist die Erfassung der Benutzungsoberflächen. Wie bereits in Abschnitt 1 angeführt, wird eine 1:1-Überführung nicht verfolgt. Vielmehr sollen

die wesentlichen Informationen erfasst und in der Zielplattform umgesetzt werden. In unserem Fallbeispiel konnten wir feststellen, dass in der Applikation drei Typen von Benutzungsoberflächen existieren. Diese stellten entweder die Funktionalität zum Anzeigen, Selektieren oder Editieren von Datensätzen bereit. Für jeden Typ wurde daraufhin eine Umsetzung in der Zielplattform festgelegt. Solche applikationsspezifischen Aspekte zu berücksichtigen soll in Form einer leichtgewichtigen Erweiterung, durch die Nutzung von Profilen, ermöglicht werden. Ein Aspekt kann dadurch in dem Abstraktionsschritt erfasst und in der Überführung ausgewertet werden.

Auf Grund der Freiheitsgrade der Quellplattform wird eine Vollautomatisierung des Abstraktionsschritts nicht angestrebt. Vielmehr soll ein Semi-Automatismus genutzt werden, indem Expertenwissen über die Applikation einbezogen und mit leistungsfähigen Automatismen kombiniert wird. Dies umfasst die Annotation der plattformspezifischen Modelle, indem z.B. die Klassifikation einzelner Code-Abschnitte in Bezug auf deren architektonische Zugehörigkeit angegeben wird. Anschließend können Transformationsregeln basierend auf dem plattformspezifischen Metamodell ausgedrückt werden, welche die angefügten Annotationen auswerten und die eine Transformation-Engine automatisch ausführt. Wir verfolgen dabei den Ansatz, die einzelnen Transformationsregeln modular in Form eines Katalogs zu verwalten, so dass diese in anderen Migrationsprojekten wiederverwendet werden können. Inwiefern eine Wiederverwendung gewährleistet werden kann, soll im Rahmen des Fallbeispiels evaluiert werden. Insbesondere im Kontext einer 4GL-Migration existieren jedoch seitens der Quellplattform vorgegebene Standard-Lösungen bzgl. der Umsetzung von Funktionalitäten, für welche ein Katalog bereitgestellt werden kann.

Die Modellstrukturen, die bis zu diesem Schritt in dem Repository gebildet wurden, müssen genutzt werden, um die Zielarchitektur der Applikation zu definieren. Dabei handelt es sich um eine manuelle Tätigkeit, bei der die extrahierten Informationen zur Entscheidungsfindung ausgewertet werden. Während und nach dem Entscheidungsprozess kann es notwendig sein, zusätzliche Informationen zu extrahieren. Aus diesem Grund wird der Abstraktionsschritt iterativ ausgeführt, und das Repository wird inkrementell angereichert. Die Definition einer Zielarchitektur wird unterstützt, indem das Wissen aus anderen Migrationsprojekten wiederverwendet wird. Beispielsweise soll auf einen Katalog bewährter Zielarchitekturen und Umsetzungsmöglichkeiten zurückgegriffen werden können. Die Beschreibung solcher bewährten Lösungen kann mit Automatismen einhergehen, welche die bestehende Applikation auf die Konformität überprüfen und Probleme aufdecken.

Aus der Zielarchitektur geht letztendlich der Umfang hervor, in welchem eine Applikation auf dieser Abstraktionsebene beschrieben wird. Die Zielplattform stellt z.B. mehrere Möglichkeiten bereit, Geschäftsregeln abzubilden. Diese können programmatisch umgesetzt oder explizit durch eine entsprechende Engine ausgewertet werden. Wenn Letzteres genutzt werden soll, müssen die Geschäftsregeln jedoch aus dem Quellcode der Quellapplikation extrahiert und in einer zugehörigen Beschreibungssprache ausgedrückt werden, während bei einer programmatischen Umsetzung Code-Wrapping eingesetzt werden könnte. Die Extraktion erzeugt in diesem Beispiel zusätzlichen Aufwand, bietet jedoch auch viele Vorteile wie eine einheitliche Verwaltung oder verständliche Repräsentation der Regeln. Allgemein stellen Modelle auf dieser Abstraktionsebene einen

Mehrwert über die Migration hinaus dar, da sie Teilaspekte der Applikation verständlich dokumentieren.

3. *Restrukturierung*
   Nachdem die aktuelle Ausprägung der Applikation auf verschiedenen Abstraktionsebenen erfasst wurde, ist der Mehrwert einer Restrukturierung der Applikation zu bestimmen, durch welche diese an die Konzepte der Zielplattform angepasst werden könnte (*Platform Independent Model*). Dies kann Anhand des plattformspezifischen Metamodells der Quell- und der Zielplattform erläutert werden, dargestellt in Abbildung 2. In Oracle Forms werden Quellartefakte zu Modulen (*FormModule*) zusammengefasst, die eine wiederverwendbare Einheit bilden. Oftmals wurden durch ein Modul Benutzungsoberflächen (*Window*) zusammengefasst, die auf ähnlichen Datensätzen arbeiten (*Block*), da die Anbindung an die Datenschicht nicht modularisiert werden konnte. In Oracle ADF hingegen ist die Modularisierung der Anbindung an die Datenschicht explizit vorgesehen (*Application Module*). Hierdurch wird die Unterteilung einer Applikation in wiederverwendbare Arbeitsschritte (*Activity*) und Arbeitsabläufe (*Task Flow*) ermöglicht. In der Migration muss evaluiert werden, inwiefern die Struktur der zu migrierenden Applikation auf Konzepte der Zielplattform abgebildet werden kann. Falls die resultierende Zielapplikation nicht ausreichend gut strukturiert ist, so dass ein Mehraufwand durch eine anschließende Restrukturierung entstünde, sollte eine systematische Restrukturierung in der Migration durchgeführt werden. Dazu ist semi-automatisch zu erfassen, welche Benutzungsoberflächen in einem Arbeitsablauf beteiligt sind. Basierend auf den resultierenden Modellen muss eine Entscheidung getroffen werden, an welchen Stellen wiederverwendbare Komponenten, entsprechend den Konzepten der Zielplattform, gebildet werden können. Der Umfang dieses Schritts variiert folglich in Abhängigkeit von der Applikation, im optimalen Fall wird er nicht benötigt.

4. *Konkretisierung*
   Aus der abstrakten Beschreibung der Applikation muss ein plattformspezifisches Modell abgeleitet werden (*Platform Specific Model*). Dazu wird für die Zielplattform ein Metamodell bereitgestellt, durch welches Applikationen plattformspezifisch beschrieben werden können. Neben den Sprachelementen der Zielplattform enthält das Metamodell zusätzliche Elemente, um bewährte Lösungsmöglichkeiten auf einer höheren Abstraktionsebene zu beschreiben. Die Ableitung des plattformspezifischen Modells wird äquivalent zum Abstraktionsschritt durch Modelltransformationen durchgeführt, wobei die einzelnen Transformationen modular in Form eines Katalogs zu verwalten sind. Dadurch soll die Wiederverwendung von Transformationen ermöglicht werden.
   Wie in Abbildung 1 angedeutet, gehen wir davon aus, dass eine Ableitung des plattformspezifischen Modells der Zielapplikation (*Platform Specific Model*) nicht nur durch Transformationen des abstrakten, plattformunabhängigen Modells (*Platform Independent Model*) erfolgt. Denn in diesem Modell wurde von technischen Implementierungsdetails abstrahiert. Solche Details sind jedoch essenziell, um vollständig lauffähigen Code zu erzeugen. Aus diesem Grund sollen in der Generierung des plattformspezifischen Modells (*Platform Specific Model*) zusätzlich Transformationen ermöglicht werden, welche Informationen des plattformspezifischen Modells

der Altanwendung (*Platform Specific Legacy Model*) aufgreifen, um einzelne Teilbereiche der Applikation automatisch und möglichst vollständig zu generieren.

5. *Code-Generierung*
   Im letzten Schritt wird aus dem plattformspezifischen Modell der Zielapplikation (*Platform Specific Model*) Quellcode (*Source*) in der Zielplattform generiert. Dazu wird ein Code-Generator verwendet, der auf dem plattformspezifischen Metamodell der Zielplattform aufsetzt.

Wir haben bereits begonnen, Teilaspekte des Ansatzes zu implementieren. Dies umfasst insbesondere die Erstellung der plattformspezifischen Metamodelle sowie eine Routine, welche die initiale Modellbildung konform zu dem definierten Metamodell durchführen kann. Dazu verwenden wir das Eclipse Modeling Framework[2] (EMF). Die Erstellung des Metamodells ist insbesondere im Kontext einer 4GL-Migration eine Herausforderung, da die meisten Strukturen proprietär sind. In unserem Fallbeispiel haben wir die Strukturen der Quellartefakte automatisch aus einem Hersteller-Werkzeug extrahiert. Zudem muss, wie in Abschnitt 3 beschrieben, das nicht explizit sichtbare Verhalten der Quellplattform beschrieben werden. Dies umfasst beispielsweise Aufrufreihenfolgen oder die Klassifizierung von plattformspezifischen Aufrufen in Code-Abschnitten.
Ein Aspekt, der von Anfang an beachtet werden muss, ist die Skalierbarkeit der entstehenden Modelle. Da die von uns betrachteten Applikationen mehrere Millionen Zeilen Code haben, müssen die Modelle effizient verwaltet werden. Dieses Problem haben wir durch die Verwendung des Eclipse CDO-Projekts[3] (Connected Data Objects) gelöst, wodurch die Modelle in einer Datenbank persistent gespeichert werden. Der zugehörige CDO-Server ermöglicht eine effiziente Abfragemöglichkeit der Modellstrukturen. Gleichzeitig wird ein verteilter Zugriff gewährleistet.
Des Weiteren sollte von Anfang eine Anbindung der Entwicklungsumgebung der Zielplattform an das Repository eingeplant werden. Manuelle Implementierungsschritte im Anschluss an die Migration können dadurch unterstützt werden, indem die Informationen der Modelle in der Entwicklungsumgebung angezeigt bzw. verwendet werden können. In unserem Fallbeispiel haben wir eine Bibliothek entwickelt, die durch die Entwicklungsumgebung der Zielplattform eingebunden werden kann. Dadurch ist ein Zugriff auf die Modellstrukturen möglich.

4 Verwandte Arbeiten

Die modellgetriebene Softwaremigration ist mittlerweile ein etabliertes Themenfeld in der Softwaretechnik. Die OMG hat aus diesem Anlass die Architecture-Driven Modernization (ADM) Task Force[4] geschaffen, um die Modellstrukturen zu standardisieren, die während des Modernisierungsprozesses entstehen und um somit die Interoperabilität von Werkzeugen zu gewährleisten. Ein Ergebnis ist das Knowledge Discovery Metamodel (KDM) [Om11], welches eine Ontologie zur Beschreibung von Aspekten eines Soft-

[2] http://www.eclipse.org/modeling/emf/
[3] http://wiki.eclipse.org/CDO
[4] http://adm.omg.org/

waresystems auf unterschiedlichen Abstraktionsebenen darstellt.
Im Projekt SOAMIG [Wi11a] wurde die Migration von Altsystemen in serviceorientierte Architekturen (SOA) untersucht, und es wurde ein Referenzprozess vorgestellt, welcher die Phasen und Disziplinen einer SOA-Migration beschreibt [Wi11b]. In diesem Kontext wurden Techniken eingeführt, um separat erstellte Modelle durch dynamische Analysen automatisch mit Code-Abschnitten der Altanwendung zu verknüpfen [FHR10].
Im Projekt DynaMod [Ho11]. werden statische und dynamische Analysen eingesetzt, um die Architektur der bestehenden Applikation zu rekonstruieren. Basierend auf diesen Informationen wird eine Modernisierung durchgeführt.
In [RGD06] wird ein Reverse-Engineering-Ansatz beschrieben, durch den aus einer bestehenden Applikation Modelle im Sinne des MDA-Ansatzes der OMG extrahiert werden. Der Quellcode der Quellapplikation wird durch Parser zerlegt und anschließend in eine plattformunabhängige Repräsentation überführt. Aus dieser Repräsentation werden UML-Modelle abgeleitet, die zur Dokumentation oder Code-Generierung genutzt werden. Die Wiederverwendung des entstandenen Migrationswissens wird jedoch nicht diskutiert.
Der Ansatz in [Fl07] ähnelt dem, den wir in unserem Projekt verfolgen. Die Quellartefakte werden ebenfalls zerlegt, in einer plattformunabhängigen Beschreibungssprache repräsentiert und letztendlich zur Code-Generierung genutzt. Dabei wird explizit auf die Wiederverwendbarkeit und Automatisierung einzelner Teilschritte eingegangen. Allerdings wird keine Restrukturierung der bestehenden Applikation berücksichtigt.
In [GC12] werden abhängige Transformationen im Kontext der Modernisierung nach dem Hufeisenmodell diskutiert. Diesbezüglich wird eine implizite und explizite Parametrisierung unterschieden.

5 Zusammenfassung und Ausblick

Im Rahmen unseres Forschungsprojekts untersuchen wir die Anwendung von Techniken der modellgetriebenen Softwareentwicklung im Kontext der Softwaremigration. Dabei betrachten wir als konkretes Fallbeispiel die Migration von Anwendungen der Plattform Oracle Forms nach Oracle ADF. In diesem Beitrag sind wir auf die Charakteristiken dieser Migration eingegangen und haben darauf aufbauend die Anforderungen an eine Werkzeugunterstützung aufgestellt, wobei die Erfahrungen mit bereits verfügbaren Lösungen eingeflossen sind.
Aufbauend auf den beschriebenen Anforderungen wurde in diesem Beitrag ein Migrationsprozess vorgestellt. Die zu Grunde liegende Idee ist die Nutzung von Modellen auf verschiedenen Abstraktionsebenen, welche durch Modelltransformationen ineinander überführt werden. Ein zentraler Aspekt ist die Nutzung einer einheitlichen Beschreibungssprache, um das in einer Migration entstehende Wissen formal festzuhalten. Die Wiederverwendung dieses Wissens unterstützt durch Automatismen soll letztendlich das wirtschaftliche Risiko in der Migration von Applikationen verringern, indem Implementierungsfehler minimiert und Kosten gespart werden.
Die technische Umsetzung des beschriebenen Ansatzes wurde bereits begonnen, insbesondere auf plattformspezifischer Ebene konnten bereits Metamodelle, eine Modellbildungs-Routine und ein Code-Generator umgesetzt werden. Im nächsten Schritt fokussieren wir die Erstellung einer plattformunabhängigen Beschreibungssprache, um eine

Abstraktion von der technischen Umsetzung erreichen zu können. Dabei berücksichtigen wir bestehende Ansätze wie KDM, ziehen aber auch explizit Erfahrungen mit ein, die im Rahmen von Migrationen im Kontext des Fallbeispiels gemacht wurden. Die Herausforderung ist, die Aspekte zu identifizieren, welche für eine Migration im Kontext des Fallbeispiels essenziell erfasst werden müssen. Des Weiteren evaluieren wir verfügbare Modelltransformationssprachen und -engines. Transformationen sollen effizient sein und inkrementell durchgeführt werden können. Zusätzlich ist eine Modularisierung von Transformationsregeln erforderlich.

Literaturverzeichnis

[SWH10] Sneed, H. M.; Wolf, E.; Heilmann, H.;: Software-Migration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung, dpunkt Verlag 2010.

[Or12] Oracle: Oracle Application Development Tools Statement of Direction: Oracle Forms, Oracle Reports and Oracle Designer. http://www.oracle.com/technetwork/issue-archive/2010/toolssod-3-129969.pdf, Accessed: 21-Dec-2012

[GGS12] Grieger, M.; Güldali, B.; Sauer, S.: Sichern der Zukunftsfähigkeit bei der Migration von Legacy-Systemen durch modellgetriebene Softwareentwicklung. Proceedings of the 14th Workshop Software-Reengineering (WSR 2012). Softwaretechnik-Trends, vol. 32, no. 2, pp. 37–38, 2012.

[KWC98] Kazman, R.; Woods, S. G.; Carrière, S. J.: Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. Proceedings of WCRE 98, pp. 154–163, 1998.

[ZGW10] Zillmann, C.; Gringel, P.; Winter, A.: Iterative Zielarchitekturdefinition in SOAMIG. Softwaretechnik-Trends, vol. 30, no.2, pp. 39–40, 2010.

[Om11] OMG, The OMG Knowledge Discovery Metamodel (KDM), Sep. 2011.

[Wi11a] Winter, A. et al.: SOAMIG Project: Model-Driven Software Migration Towards Service-Oriented Architectures. Proceedings of MDSM 2011, vol. 708 of CEUR Workshop Proceedings, p. 15–16, 2011.

[Wi11b] Winter, A. et al.: The SOAMIG Process Model in Industrial Applications. Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR), pp. 339–342, 2011.

[FHR10] Fuhr, A.; Horn, T.; Riediger, V.: Dynamic Analysis for Model Integration. Softwaretechnik-Trends, vol. 30, no. 2, p. 70–71, 2010.

[Ho11] van Hoorn, A. et al.: DynaMod Project: Dynamic Analysis for Model-driven Software Modernization. Proceedings of MDSM 2011, vol. 708 of CEUR Workshop Proceedings, pp. 12–13, 2011.

[RGD06] Reus, T.; Geers, H.; Van Deursen, A.: Harvesting Software Systems for MDA-based Reengineering. In: Model Driven Architecture-Foundations and Applications, pp. 213–225, 2006.

[Fl07] Fleurey, F. et al.: Model-driven Engineering for Software Migration in a Large Industrial Context. In: Model Driven Engineering Languages and Systems, pp. 482–497, 2007.

[GC12] Cuadrado, J. S.; García, O. Á.; Canovas, J.; Herrera, A. S. B.: Parametrización de las transformaciones horizontales en el modelo de herradura. Jornadas de Ingeniería del Software y Bases de Datos, 2012.

Future Research Topics in Enterprise Architectures Evolution Analysis

Sascha Roth, Florian Matthes

Software Engineering betrieblicher Informationssysteme (sebis)
Technische Universität München
Boltzmannstr. 3, 85748 Garching bei München
roth@tum.de, matthes@in.tum.de

Abstract: An Enterprise Architecture (EA) embraces an organization's technical infrastructure, applications, business capabilities, and relationships among them. Evolutionary design is a common characteristic for such an EA. An EA can be considered as a complex system of systems, whereas the actual efforts for its maintainability and evolution have a high influence on the enterprise's capability to quickly respond to market changes. A common means to analyze an EA are visualizations. However, research on visual means for analysis of EA evolutions is scarce. In this paper we outline visual means to analyze the evolution of EAs and motivate research on this topic by outlining related challenges.

1 Introduction and Motivation

Over the past decades enterprise application landscapes have grown to complex systems of systems intrinsically complex and hard to manage as a whole [WR09]. As a reaction, Enterprise Architecture (EA) management seeks to cope with this complexity. Goal of this young discipline is to increase business/IT alignment, while reducing operational costs in order to respond to frequently changing business models, to enable faster business transformations, and, thus, increase shareholder return. Commonly, an EA endeavor starts by documenting the current (AS-IS) state in an EA repository. Especially when focusing on recent trends [BE12] such as *automated EA documentation* (cf. [Bu12,Gr12,Fa13,Ro13]) the underlying information model (cf. [Le99]) of such an EA repository tends to change frequently. Automated EA documentation consolidates existing EA information sources from an operative IT environment in order to gather relevant information. Practical applications show, that for automated EA documentation, the EA repository's information model is extended over time [Ha12b], i.e. it evolves.

According to [Bu08], EA visualizations[1] consist of a finite number of variations identified as EA patterns, such that they can be predefined in an abstract manner by employing variability points [SMR12,Ha12a]. We implemented these visualizations

[1] We refer the interested reader to [Bu08] for a predefined set of best-practice EA visualizations gathered from industry.

based on a non-rigid typed system called Hybrid wiki [Ma11]. This web-based wiki system allows capturing not only plain text, but also structured information in a user-friendly manner. Moreover, it can be used as a flexible store for automated EA documentation [Ha12b]. In [Bu10b] we sketched a conceptual model where we outlined the evolutionary characteristic of EA transformations guided by principles and standards. Thereby, also agile approaches, e.g. [Bu11], could be applied. With respect to agile approaches that may change EA information more frequently and against the background of the recent trends of automated EA documentation, an overview of the EA evolution could be highly beneficial for an evidence-based management of the EA.

EA management has to serve information to multiple stakeholders with different concerns and often unclear goals. Due to the diverse nature of these concerns, stakeholders require their individual viewpoints (cf. [ISO07]). Large enterprises commonly comprise 10^2-10^3 information systems interconnected via interfaces, i.e. respective viewpoints represent complex interlinked data such that visualizations (views) have been established as a common means for analysis [Ha12a].

As of today, visualizations can be generated based on EA models applying transformation techniques [Ha12a,SMR12]. However, when it comes to historical information of EAs, i.e. analyzing its evolution, literature is scarce. In this paper we outline a brief example for a fictitious enterprise to illustrate challenges and the complexity EA management has to deal with. Our long-term research question is: *How can we visualize the evolution of an EA model?* This paper serves as a basis for discussion and problem understanding.

2 Analyzing the Evolution of Enterprise Architectures

In our example (cf. Figure 1), a fictitious enterprise has been grown by Mergers & Acquisitions (M&A). For the application landscape consolidation, they chose an absorption strategy (cf. [Ec12]). In this particular scenario, a new subsidiary in Hamburg has been acquired. The respective applications of acquired (and mostly redundant) business units have to be migrated in terms of customer data and/or entire application systems. The former often represents the real asset acquired through an M&A. However, for simplicity, we abstract from that.

Figure 1 illustrates multiple migration steps beginning with the acquisition in 2012 until a final state in 2016. Until then, acquired business units should be entirely absorbed by the existing ones. Applications of a consolidated business unit are not used, but fully functional for the purpose of proper data migration. In 2013, the enterprise plans to migrate applications of the IT department located in Hamburg to the applications of the IT department located in Berlin. In 2014, most of the applications of the R&D unit located in Hamburg should be absorbed by the R&D unit located in Frankfurt. In 2015, remaining applications of HR, production, and R&D located in Hamburg should be absorbed by the respective facilities in New York.

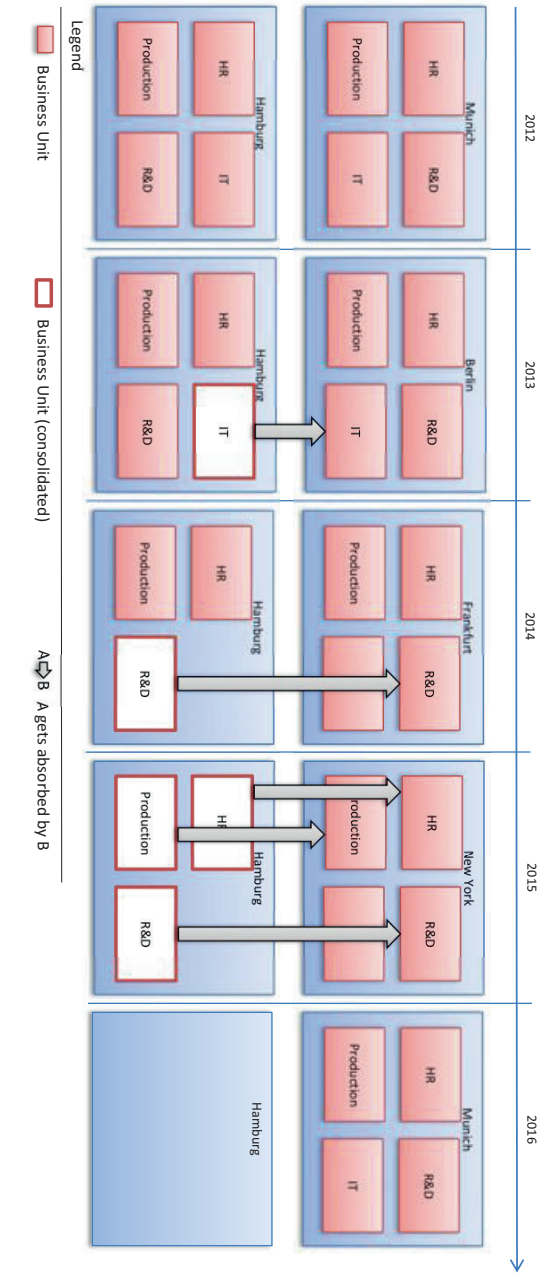


Figure 1: Visualization of multiple migration steps of an EA during an M&A scenario.

In Figure 1, we applied time series [Tu01] to the domain of software cartography [Ma08]. The boxes are ordered by the transformation steps. Note that the outer boxes stay at the same size, such that existing box layout algorithms, e.g. the next-fit decreasing height (NFDH) algorithm, cannot be applied to sort and layout visual elements.

3 Limitations, Challenges, and Ideas for Future Research

Against the background of automated EA documentation, we assume that increasingly operative information will be imported in EA tools. Thus, the amount of data will increase and for planning purposes must be compared with existing 1) previous data, 2) planned states, and 3) a target state. During implementation, we identified the following challenges that arise when visualizing the evolution of EAs:

Scalability. The presented solution can only be applied when comparing two entities (business units) with each other. We identify the arrangement of multiple business units, or more general multiple entities becomes an issue when applying existing EA viewpoints to historical information. Viewing at more than one dimension at a time easily ends up in an information overflow. Different visualizations known from EA management must be re-evaluated, whether they might serve for the visualization of evolutionary information, e.g. binary/ternary matrix, (directed acyclic) graph, (cf. e.g. [Bu08]).

Scoping vs. "Big Picture". Other scenarios often seen in the domain of EA management are standardization of business applications, harmonization, etc. (cf. also Figure 2) such that the history (versions) of one business unit or one business application could be subject of interest for a deeper analysis. In contrast, a high-level overview sometimes is also beneficial. Of course, this strongly depends on the actual stakeholder.

Layout algorithms. So far applied layout algorithms for EA visualizations aim at esthetical pleasing layouts, e.g. decreasing the overall height of boxes with respect to an aspect ratio in case of the NFDH algorithm. For instance this particular algorithm would close any open space (white-spots) that are created, e.g. during migrations and are actually useful to communicate (the amount of) change when analyzing EA evolutions.

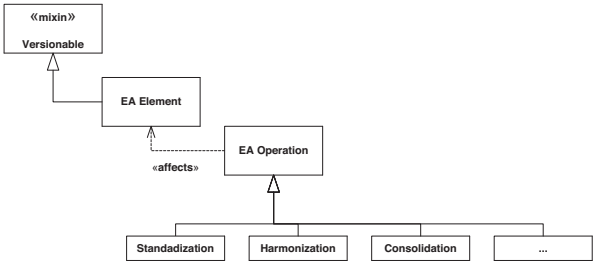


Figure 2: Meta-model to illustrate the relationship between versions, EA Element, and EA Operations

Generic vs. special purpose solutions. We expect some of the solutions developed will be generic, whereas others will be very specific for the domain of EA and EA management (similar to [Bu10a]). Figure 2 illustrates the relationship of versions of EA elements to EA operations. These operations are usually executed as methods through a project. In order to communicate change, the actual applied method must also be captured by an EA meta-model. The (meta-)data required to generate such visualizations of the EA evolution in an automated manner currently has not been addressed by existing EA literature. For instance, must transformation operations be modeled and maintained explicitly or is it possible to infer respective information from existing data or at least in an automated manner from existing information sources.

Communication of timespans. When comparing two different EA states with each other, it might be beneficial to get visual feedback on the actual timespan (and time differences) visualized. Existing visualization approaches, e.g. [Tu01], may be helpful for this purpose.

Above outlined challenges represent a non-extensive list to be addressed by further research. In a first step, this list should be evaluated concerning practical relevance in the domain of EA management.

4 Outlook

In this paper we illustrated a brief example of an organization which absorbed an entire EA in the vein of an M&A scenario. Especially when focusing on historical information (versions of the EA) we outlined several issues to be addressed by further research. The illustrated example serves to motivate the problem and its complexity rather than sketching a solution.

We seek to discuss the presented approach, other visual representations and possibilities to communicate evolution of models in general and in particular of EA models at the Design for Future 2013 workshop. In particular we foresee discussions about analogies between communication of change (and evolution) of software architecture and EA models.

References

[BE12] Berneaud M., Buckl S., Fuentes A., Matthes F., Monahov I., Nowobilska A., Roth S., Schweda C.M., Weber U., Zeiner M.: Trends for Enterprise Architecture Management and Tools. Technical Report 2012.

[Bu08] Buckl, S.; Ernst, A.; Lankes, J.; Matthes, F.: Enterprise Architecture Management Pattern Catalog (Version 1.0, February 2008). Technical Report TB0801, Chair for Informatics 19 (sebis), Technische Universität München, 2008.

[Bu10a] Buckl, S.; Matthes, F.; Roth, S.; Schulz, C.; Schweda, C. M.: A method for constructing enterprise-wide access views on business objects. In: Informatik 2010: IT-Governance in verteilten Systemen (GVS 2010), Leipzig, 2010.

[Bu10b] Buckl, S.; Matthes, F.; Roth, S.; Schulz, C.; Schweda, C.M.: A Conceptual Framework for Enterprise Architecture Design. In: Workshop Trends in Enterprise Architecture Research (TEAR 2010), Delft, 2010.

[Bu11] Buckl, S.; Matthes, F.; Monahov, I.; Roth, S.; Schulz, C.; Schweda, C.M.: Towards an agile design of the enterprise architecture management function. In: Trends in Enterprise Architecture Research (TEAR) - 6th International Workshop, Helsinki, 2011.

[Bu12] Buschle, M., Ekstedt, M., Grunow, S., Hauder, M., Matthes, F., Roth, S.: Automating Enterprise Architecture Documenation using Models of an Enterprise Service Bus. In: Americas Conference on Information Systems (AMCIS 2012), Seattle, Washington, USA, 2012.

[Ec12] Eckert, M.-L., Freitag, A., Matthes, F., Roth, S., & Schulz, C.: Decision support for selecting an application landscape integration stratetegy in mergers and acquisitions. 20th European Conference on Information Systems (ECIS 2012). Barcelona/Spain, 2012.

[Fa13] Farwick, M., Hauder, M., Roth, S., Matthes, F., Breu, R.: Enterprise Architecture Documentation: Empirical Analysis of Information Sources for Automation - In the 46th Hawaii International Conference on System Sciences (HICSS 46), Maui, Hawaii, 2013.

[Gr12] Grunow, S., Matthes, F., Roth, S.: Towards Automated Enterprise Architecture Documentation: Data Quality Aspects of SAP PI. In: 16th East-European Conference on Advances in Databases and Information Systems (ADBIS), Poznan, Poland, 2012.

[Ha12a] Hauder, M., Matthes, F., Roth, S., Schulz, C.: Generating dynamic cross-organizational process visualizations through abstract view model pattern matching, Architecture Modeling for Future Internet enabled Enterprise (AMFInE 2012), Valencia, Spain, 2012.

[Ha12b] Hauder, M., Matthes, F., Roth, S.: Challenges for Automated Enterprise Architecture Documentation. In: 7th International Workshop on Trends in Enterprise Architecture Research (TEAR), Barcelona, Spain, 2012.

[ISO07] ISO/IEC/IEEE 42010:2011, *Systems and software engineering — Architecture description*, the latest edition of the original IEEE Std 1471:2000, *Recommended Practice for Architectural Description of Software-intensive Systems*.

[Le99] Lee, Y.T. (1999) Information modeling: From design to implementation, Second World Manufacturing Congress, pp. 315–321.

[Ma08] Matthes, F.: *Softwarekartographie*. In: Informatik-Spektrum, Vol. 31, No. 6, S. 527-536, Springer-Verlag, 2008, DOI 10.1007/s00287-008-0289-2.

[Ma11] Matthes, F.; Neubert, C.; Steinhoff, A.: Hybrid Wikis: Empowering Users to Collaboratively Structure Information. In: 6th International Conference on Software and Data Technologies (ICSOFT), Pages 250-259, Seville, 2011.

[Ro13] Roth, S; Hauder, M., Farwick, M., Matthes, F., Breu, R.: Enterprise Architecture Documentation: Current Practices and Future Directions, 11th International Conference on Wirtschaftsinformatik (WI), Leipzig, Germany, 2013.

[SMR12] Schaub, M.; Matthes, F.; Roth, S.: Towards a Conceptual Framework for Interactive Enterprise Architecture Management Visualizations. In: Modellierung, Bamberg, Germany, 2012.

[Tu01] Tufte, E. R. The Visual Display of Quantitative Information. Graphics Press, 2001.

[WR09] Weill, P. and Ross J. W. (2009). IT Savvy – What Top Executives Must Know to Go from Pain to Gain, Harvard Business Press.

Evolution wiederverwendbarer Schnittstellen in der Produktentwicklung

Simon Giesecke, Niels Streekmann

BTC Business Technology Consulting AG, Oldenburg / Berlin
Geschäftseinheit Energie Produkte
simon.giesecke@btc-ag.com
niels.streekmann@btc-ag.com

Abstract: Im Rahmen der Entwicklung individualisierbarer Standardprodukte bei der BTC AG spielt die Wiederverwendung von Standardschnittstellen eine besondere Rolle. Besonders relevant ist dabei die Betrachtung von API- und ABI-Kompatibilität. Dieser Beitrag beleuchtet die dabei zu betrachteten Dimensionen sowie Strategien zum Umgang mit inkompatiblen Änderungen.

In der Geschäftseinheit Energie Produkte entwickelt die BTC AG mehrere Softwareprodukte für die Energiewirtschaft, bei denen es sich vorwiegend um individualisierbare Standardprodukte handelt. Bei der Entwicklung stützt sie sich auf die Wiederverwendung, wobei ein Schwerpunkt auf einer Reihe von Standardschnittstellen liegt [Sie04], über die flexible Optionen hinsichtlich der Ausprägung von Qualitätseigenschaften und technischen Rahmenbedingungen geschaffen werden können [FFGL11]. Dabei bedeutet Wiederverwendung hier insbesondere, dass die Einsatzkontexte von entwickelten Schnittstellen, Komponenten und Frameworks nicht vorab abschließend bekannt sind und sich diese auch in getrennten Quelltextrepositories befinden. Hierbei stellen sich für die Evolution besondere Herausforderungen:

- Berücksichtigung der Abwärtskompatibilität beim Entwurf
- Unterstützung der Ersetzung von Schnittstellen und Komponenten durch kompatible Versionen durch Provisionierungswerkzeuge

Wir unterscheiden Schnittstellen und Komponenten im komponenten-orientierten Sinne, bei denen diese getrennte Artefakte mit separatem Lebenszyklus darstellen, und Frameworks, die zwar eine Schnittstelle exportieren, welche aber an eine einzige Implementierung gebunden ist. Für Schnittstellen besonders relevant sind dabei API- und ABI-Kompatibilität (Application Programming/Binary Interface).

Für Komponenten ist die API-Kompatibilität bei korrekter Anwendung des Liskov'schen Substitutionsprinzips nicht von Bedeutung. Die ABI-Kompatibilität ist nur von sehr untergeordneter Bedeutung, da hier lediglich eine sehr schmale Schnittstelle von außen genutzt wird (Factories u.ä.). Wesentlicher sind hier Kompatibilität der Konfiguration und ggf. von persistenten Daten (Datenbankschemata, Dateiformate), Übertragungsprotokollen und ähnlichem. Für Frameworks sind sowohl API- als auch ABI-Kompatibilität relevant, und zudem von besonderer Komplexität, z.B. wegen der Verwendung offener Templates (in C++).

Weitere Kompatibilitätsaspekte (z.B. konzeptionelle Kompatibilität) werden von uns nicht näher betrachtet. Von Kompatibilität zu unterscheiden sind zudem Portabilität zwischen verschiedenen Plattformen oder die Interoperabilität zwischen verschiedenen Compilern auf derselben Plattform.

API-/ABI-Kompatibilität hat verschiedene Dimensionen:

1. Syntaktische vs. semantische Kompatibilität
2. Quelltext- vs. Binärkompatibilität
3. Strenge vs. lockere Kompatibilität
4. Client- vs. Provider-Kompatibilität

Im Zusammenhang mit der Komponenten-orientierten Entwicklung hat die letzte Dimension besondere Bedeutung, wobei Standardschnittstellen besonders kritisch sind. Clients einer Schnittstelle A sind hierbei Module, die die Schnittstelle A nutzen, Provider sind Module, die die Schnittstelle A implementieren. Bei vielen Änderungen ist es so, dass sie entweder für Clients (z.B. Entfernen einer Methode) oder für Provider (z.B. Hinzufügen einer Methode) inkompatibel sind. Aus diesem Grund erlaubt z.B. auch COM keinerlei Änderungen vorhandener Schnittstellen. Eine Standardschnittstelle, wie sie hier verstanden wird, ist aber auf einer höheren Granularitätsebene angesiedelt und entspräche einer Sammlung von COM-Schnittstellen. Hierin liegt auch der Lösungsansatz: Ein Hinzufügen einer optionalen Schnittstelle zu einer Sammlung feingranularer Schnittstellen ist sowohl für Clients als auch Provider kompatibel. Damit diese sinnvoll genutzt werden kann, muss dies jedoch bereits vor der ersten Erweiterung bedacht und geeignete Maßnahmen müssen ergriffen werden.

Der Vortrag beleuchtet die genannten Dimensionen anhand von konkreten Anwendungsszenarien und erläutert ihre Bedeutung für die Produktentwicklung auf der Basis von Standardschnittstellen:

- Szenario 1: Identifikation eines Bugs in einer Komponente oder einem Framework nach Auslieferung an einen Kunden
- Szenario 2: Anforderungen aus der Produktentwicklung, die nur durch Erweiterung einer Standardschnittstelle umgesetzt werden können

Des Weiteren werden Strategien vorgestellt, um inkompatible Änderungen mittels eines geeigneten Prozesses zu steuern, mit Unterstützung von Werkzeugen zu erkennen und sie bekannt zu machen.

Literatur

[FFGL11] Frenzel, M.; Friebe, J.; Giesecke, S.; Luhmann, T.: Standardschnittstellen als nichtfunktionale Variationspunkte: Erfahrungen aus der EPM-Produktlinie. In Reussner, R.; Pretschner, A.; Jähnichen, S. (Hrsg.): Software Engineering 2011 Workshopband, Workshop Produktlinien im Kontext: Technologie, Prozesse, Business und Organisation (PIK2011), Lecture Note in Informatics, Volume P-184, S. 253–264, 2011.

[Sie04] Siedersleben, J.: Moderne Softwarearchitektur, dpunkt Verlag, 2004.

ISS Columbus Module On-Board Software Maintenance

Jasminka Matevska

Astrium Space Transportation GmbH
Airbusallee 1
28199 Bremen
jasminka.matevska@astrium.eads.net

Abstract: Ensuring correct and consistent functionality during utilization and enhancement of long-living complex space systems as the ISS Columbus Module is one of the main challenges for its maintenance approach. The Columbus Laboratory consists of many subsystems in order to assure vital and working conditions for the on-board crew and execution of different science experiments in space. The on-board systems are functioning autonomous and are supported by software systems and control centers on ground. The utilization of the systems requires processing of a huge amount of monitoring and control data in near real time. Due to criticality of some system functions, their availability has to be sustained even during performing maintenance activities coping with different components, target platforms and responsibilities. This paper provides an overview of the ISS Columbus Module On-board Software Maintenance Process Model.

1 Introduction

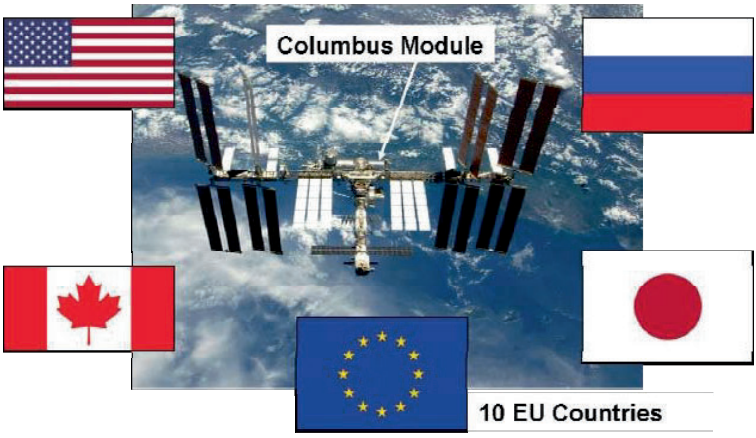


Figure 1: International Space Station (ISS)

The International Space Station (ISS) is the largest space project in history so far. It consists of different attached modules used as laboratories for science experiments in the microgravity in space. Each module provides its own system consisting of many at principle autonomous subsystems in order to assure vital and working conditions for the on-board crew. This requires processing of a huge amount of monitoring and control data in near real time.

Many countries are contributing in this project (e.g. USA, Russia, Canada, Japan, Germany, France, Italy, UK, and Spain). The European contribution to the ISS is the Columbus Module. It has been launched and attached to the International Space Station in February 2008 and shall be utilized until at least 2020. Three crew members can work and perform various science experiments on-board Columbus. In principle, the system is functioning autonomous and operated in orbit. It is commanded mainly from ground by corresponding control centers and supported by ground engineering, integration, verification and test facilities. Although Columbus is a European Module, there are many interfaces both on system and organizational level with the international partners. Monitoring, operation and commanding of the different subsystems (e.g. water pump, electrical system, data management system) is supported by communicating software systems on-board and on ground. Therefore, the Columbus Module can be considered a complex long-living space system both from the technical as well as from the organizational point of view.

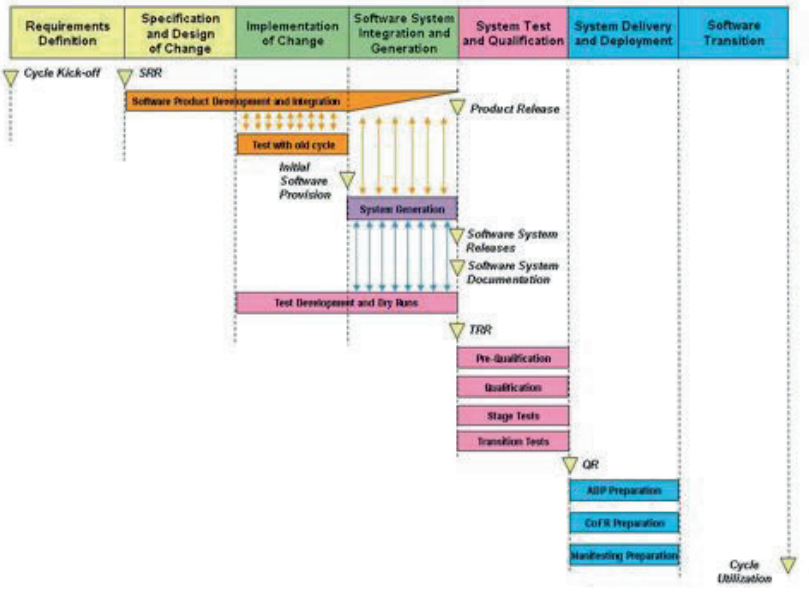


Figure 2: Columbus Module Software Cycle Approach

Due to the complexity of the system and its interfaces, it is a considerably big challenge to ensure a well functioning and consistent on-board system and supporting ground facilities over a long period of time.

During the utilization phase of Columbus, there is a necessity of efficient maintenance of all Columbus subsystems and thus maintenance of the on-board software system.

Maintenance of the Columbus on-board software system includes all types of maintenance: corrective, adaptive, perfective, and preventive. It follows the software cycle approach (Figure 2). Each cycle can be understood as a software development iteration embedded within the V-Engineering Process Model [PRD]. Due to high availability and reliability requirements of some subsystems, the preparation of a new cycle has to be performed in parallel to sustaining activities of the flight following cycle (the one installed and utilized at present). In principle, a cycle is following the phases and milestones of software development as defined in the ECSS standards (European Cooperation for Space Standardization). Included steps are described as follows.

2 Requirements Definition

All requirements of the utilized on-board software system are documented within a maintenance specification document [MSP]. All problems and enhancement requests identified during utilization are tracked via a dedicated problem tracking tool [SPR] and analyzed by corresponding boards and responsible engineers. At the beginning of one software cycle the baseline definition of the outstanding cycle is agreed and the delta of problems and enhancement requests to be implemented is identified (Baseline Kick-Off Review). According to the analysis results, additional requirements are defined or existing requirements are revised (System Requirements Review). Additionally, the verification method of all requirements is determined [MSP, VCD].

3 Specification and Design of Changes

All identified changes are analyzed and the impact to the system and corresponding system documentation is defined. All specification, design or interface changes are documented in the appropriate architecture definition and interface specification documents [ADD, ICD].

If any software interfaces to the international partners (e.g. NASA) are impacted, the impacted ISS documents are updated [ISS] and the corresponding NASA/ESA multilateral boards have to approve the changes.

Since the system is already utilized and only design changes are specified, this phase is considered completed by a Delta Design Review.

4 Implementation of Changes

After successful definition of requested changes, the detailed design of changes is defined as a foundation for the corresponding implementation work.

All impacted products are updated or extended by the responsible engineer and provided to the responsible board for release.

5 Integration and Generation of Software System

All released products are integrated in a software system assembly by a dedicated in-house developed generation environment [GEN, IWB]. The generation approach has to cope with a heterogeneous system composed of multiple software components which differ in terms of responsible organization, target platforms, and development environments. The generation environment provides all necessary assemblies of the on-board software system and all variants for ground facilities.

Nevertheless, the automatic generation of the software, release tracking, configuration management of versions and variants is not the focus of this paper, since not in our responsibility.

6 System Test and Qualification

Qualification of Columbus on-board software includes different verification methods: I (inspection), ROD (review of design), T (test) or A (analysis) [MSP, VCD].

Closeout of all requirements having "T" as verification method assigned is provided by tests on different test levels.

Product level tests are performed as part of the product change implementation.

For system level tests, multiple test facilities are used:

- Software test facility including a software simulation of electrical systems
- Flight following test facility including flight hardware
- Training, qualification and verification facility
- ISS system verification facility

As preparation for tests, all necessary test procedures are created or updated according to the change definition [TED].

Before starting with the official qualification phase, the Dry-Run test phase is performed. This phase is used to finalize the system configuration and test procedures. Identified

problems can be fixed by performing internal iterations consisting of steps as described in the sections 4, 5 and 6 (Implementation, Integration and Test).

The mandatory milestone in order to start the formal qualification phase is the Test Readiness Review. At this time point, the complete software system including test procedures and facilities shall be integrated and released. It is essential to assure a well defined configuration both of the system under test and all test facilities.

The first phase of the software system qualification campaign is called Pre-Qualification and is performed on the software test facility including a software simulation of electrical systems. The Columbus software system team aims to perform as many tests as possible on this facility in order to discover problems and provide fixes as early as possible. Problems discovered during this phase can be fixed by patching the system assemblies.

After successful pre-qualification the succeeding tests are performed on the flight following facility (Qualification) and the ISS system verification facility (Stage Tests).

For dedicated qualification of operational products and training of astronauts, the training facility is used.

The Columbus on-board software system is considered qualified for the defined cycle after successful closure of all requirements. The final milestone is the Qualification Review.

7 System Delivery and Deployment

The qualified Columbus on-board system is released and delivered for on-board deployment following the complete process of final integration, generation and release of the software.

For on-board delivery, the necessary preparation for flight is performed. All items have to provide a Certificate of Flight Readiness (CoFR) including a variety of material test reports and safety assessments (Acceptance Data Package (ADP)). For final approval for flight, all items are packed and documented within appropriate manifesting documents. After approval by the Cargo Review for the assigned spacecraft vehicle (e.g. Progress, Soyuz), they can be deployed to orbit.

8 Software Transition

On-board, the data carrier of the on-board system software is used for installation of the software. The installation of the qualified software is performed as a stepwise software transition from one software cycle to another in order to assure the availability of critical system functions. Due to technical complexity of the system and organizational complexity of supporting engineering and control centers, additional operational

preparations are done. All operational aspects are defined, documented and released in appropriate procedures [OPS]. A dedicated transition procedure is defined and transition tests are performed. Finally, the Transition Readiness Review approves the system readiness for transition.

After successful transition the utilization phase for the new software cycle can start.

9 Summary

The ISS Columbus Module is successfully utilized since its launch and attachment to the ISS in February 2008. This success is a result of efficient and high quality work of various teams on-board ISS and on ground in the supporting engineering and control centers. The cycle approach for maintenance of the Columbus on-board software system has been followed with success for many years starting with the activation cycle 10.1 as the first utilization software system cycle. At present, the cycle 13 is utilized on-board. The qualification phase of cycle 14 is the ongoing activity on ground.

References

[ADD12] Columbus On-board Software Architectural Design Documentation, Astrium Space Transportation Internal Documentation, 1998-2012

[ICD12] Columbus On-board Software Interface Control Documentation, Astrium Space Transportation Internal Documentation, 1998-2012

[PRD12] Columbus On-board Software Maintenance Process Documentation, Astrium Space Transportation Internal Documentation, 1998-2012

[MSP13] Columbus On-board Software Maintenance Specification, Astrium Space Transportation Internal Documentation, 2013

[VCD13] Columbus On-board Software Verification Control Document, Astrium Space Transportation Internal Documentation, 1998-2013

[TED13] Columbus On-board Software Test Documentation, Astrium Space Transportation Internal Documentation, 1998-2013

[SPR13] Columbus System Problem Report Database Documentation, Astrium Space Transportation Internal Documentation, 1998-2013

[OPS13] Columbus Operational Documentation, ESA/DLR/Astrium Space Transportation Internal Documentation, 1998-2013

[ISS12] NASA International Space Station Documentation, NASA/ESA Internal Documentation, 1998-2012

[GEN13] Columbus Software Generation Documentation, ESA/DLR/Astrium Space Transportation Internal Documentation, 1998-2013

[IWB10] Ignatova, T.; Westerholt, U.; Brandt, M.: Advanced Software Maintenance Approach for the Complex Columbus Flight Software System. Proc. of Data systems in Aerospace (DASIA) 2010, Budapest, Hungary, 2010; P. 289-292.

[ECSS12] ECSS Standards, European Cooperation for Space Standardization, 1993-2012

Towards identifying evolution smells in Software Product Lines

Klaus Schmid[1], Rainer Koschke[2], Christian Kröher[1], Dierk Lüdemann[2]

[1]University of Hildesheim, Institute of Computer Science, Software Systems Engineering, Marienburger Platz 22, 31141 Hildesheim, Germany
{schmid|kroeher}@sse.uni-hildesheim.de

[2]University of Bremen, Institute of Computer Science, Softwaretechnik, Am Fallturm 1, 28359 Bremen
koschke@informatik.uni-bremen.de, dierk@tzi.de

Abstract: As more and more companies shift to a product line approach, supporting the evolution of software product lines becomes increasingly important. While today already significant work exists along the lines of quality analysis for software product lines, there is much less work that addresses the evolution scenario. In this paper, we briefly describe different categories of approaches for identifying problems in product lines. Based on this we describe a new research direction for identifying problems in product line evolution scenarios.

1 Motivation

Software evolution is known to be a difficult problem in software engineering. Often errors are introduced into the software as part of the evolution; the difficulty of evolution is compounded by the fact that developing an increment often requires modifying many different parts of the software in a well-coordinated fashion. Any mistake in this may lead to defects. While this is well-known to be a problem in traditional single-system development, it is even more of a problem for software product lines [Sc09, LSR07]. Three reasons lead to the increased complexity in this situation:

- Product lines are longer lived and evolve as long as any of their constituent products evolve.
- Modifications will typically impact a range of products, but not all products. However, the impact on individual products is often not clear when making changes. Thus, what amounts to a correction for one product, may introduce a defect into another one.
- A product line encompasses more artifacts of a larger size than any individual product. Moreover, a product line will typically contain a variability model in addition to the other typical development artifacts [SJ04].

All these issues taken together emphasize the need for supporting the evolution of software product lines in a way that reduces the probability of defects. Unfortunately, today there exists only limited support for identifying problems introduced in product line evolution. In this paper, we will briefly outline some ways to identify problems that are introduced as part of product line evolution.[1]

Further, we will make the assumption in this paper that a product line has already been established. We will not discuss the problem of identifying, for example, copy-and-paste reuse and integrating the corresponding variants into a product line [KFBA09].

2 Approaches to Analyze Defects

Our focus in problem identification is in particular on the (semi-)automatic detection of problems that may exist or be introduced in product lines. This direction of research is, of course, not new. Analysis of product lines has been extensively discussed in the literature [BSR10], however, mostly not with a focus on analyzing evolution, but rather with a focus on scrutinizing a single state of a product line.

From this starting point, we can identify three major directions relevant to analyzing quality problems in product lines.

Variability Model Problems

The first category of approaches is to simply analyze the variability model for any obvious problems. This kind of research has significant history in the field of product lines [BSR10]. Some examples of such an approach are

- Analyze any inconsistencies in the variability model (this would imply that no consistent product may be derived)
- Dead feature analysis (a dead feature is a feature, which can never be selected as this would lead to a contradiction)

The key characteristic of such an approach is that it analyzes only the variability model without taking the realization artifacts of the product line into account at all.

Variability Coordination Problems

However, the variability model is not the only source of variability information. Rather the variability realization, i.e., the code may contain variability information as well. For example, an implementation using preprocessors like the C-preprocessor (which will also be of prime interest to the EvoLine-project) may imply dependencies among

[1] Based on this motivation and along the ideas outlined in this paper, the EvoLine-Project was created, which is part of the German Research Foundation (DFG) Priority Programme SPP 1593.

variabilities [SLBWC11]. If an explicit variability model exists, the dependencies should conform to the variability model. Moreover, we can perform the same analysis, which we discussed in the previous section as well on this implied variability model.

Where do these implications come from? Imagine, for example, that conditional compilation is used realizing two features A and B. However, each condition for B is contained within code, which is conditional on A. Thus, we can derive an implied dependency: B→A. Besides the preprocessor code also the build system, i.e., make-files, may be the source of dependencies.

If on the other hand the variability model permits a configuration B∧¬A we know that variability model and code are mutually inconsistent. While we cannot say which one is correct, we know there is a underlying problem either in the variability model or in the implemation that should be identified and corrected.

Semantic Variability Problems

While the two categories above relied on a simple analysis of variability model information, we can easily go further and analyze semantic information from the code more deeply. Some examples of such an approach are

- *Type inconsistencies*: pre-processor code may even modify type information. This can lead to situations where most, but not all, instances of the product line are type-correct. [KGREOB11]

- *Data-flow issues*: similarly, code manipulation may lead to some instances where necessary variables are not initialized. [RQBTBS11]

There are many examples of these kinds of analysis. Effectively all analysis approaches for single system code that help to detect code smells can be lifted to the product line situation. Of course, the introduction of the product line basis leads to a combinatorial explosion. This turns even a problem like determining type-correctness, which is not regarded as a major challenge for single systems, into a complex problem that also requires significant computation time.

3 Evolutionary Analysis

The problems described above have already been studied to a significant degree in product line engineering. However, these types of analysis have typically not been made in an evolution context. If we take the perspective of an evolution problem, the description changes slightly. Instead of asking whether a certain model is, for example, consistent, we ask whether an atomic change (typically a change as described as a change set in a commit-operation) introduces a problem.

On the one hand, this makes the situation more complex as it increases the amount of information that is available and potentially needs to be taken into account even further (besides the product line information also the change set).

On the other hand, we can decide to focus on the change itself. That is, we assume that the product line was correct before the change and we are only interested in the change itself. We can phrase this problem as:

If we assume the product line PL was initially correct and consistent, and we perform a change C, yielding a product line PL', can we then deduce that PL' is consistent and correct? Thus, instead of asking:

correct(PL'), we ask for *correct*(PL) → *correct*(PL').

As an individual change set is typically much smaller than the product line, we can assume that this will usually lead to significantly reducing the amount of information that needs to be taken into account. However, this amount of information will typically be significantly larger than the change set C alone, as it needs to be interpreted in context.

All taken together, we regard it as a valid research hypothesis that the evolutionary approach will actually lead to significantly more efficient decision procedures to identify any potential evolution issues (evolution smells) in the product line. As positive side-effect this will focus the feedback only on those issues that were introduced in the last iteration, thus providing directly relevant feedback.

4 Conclusion

In this paper, we introduced the concept of *product line evolution smells*, which extends existing product line analysis approaches to the evolution case. We also showed how we expect to optimize existing analyses by making them more efficient through incremental semantics. We expect the evolutionary approach to provide faster results to the developer, thus making the analysis more useful. We also aim at making the results more relevant by focusing on those parts that have recently changed. However, the concepts described above are the basis for research in progress. Thus, whether these advantages can actually be realized is currently open and will be studied further in the EvoLine-project.

5 Aknowledgements

This work was partially supported by the EvoLine-Project, funded by DFG within the SPP1593. Any opinions expressed herein are solely by the authors and not of the DFG.

References

[BSR10] Benavides, D.; Segura, S. & Ruiz-Cortes, A. Automated Analysis of Feature Models 20 Years Later: A Literature Review, Information Systems, 2010, Vol. 35, No.6, 615-636.

[KFBA09] Koschke, R.; Frenzel, P.; Breu, A. & Angstmann, K. *Extending the Reflexion Method for Consolidating Software Variants into Product Lines*, Software Quality Journal, Vol. 17, No. 4, pp. 331–366, 2009.

[KGREOB11] Kästner, C.; Giarrusso, P.; Rendel, T.; Erdweg, S.; Ostermann, K. & Berger, T. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA),* 805-824, 2011.

[LSR07] van der Linden, F.; Schmid, K.; Rommes, E.; Product Lines in Action, Springer, 2007.

[RQBTBS11] Ribeiro, M.; Queiroz, F.; Borba, P.; Toledo, P.; Brabrand, C. & Soares, S. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *International Conference on Generative Programming and Component Engineering*. pp. 23-32, 2011.

[Sc09] Schmid, K.: Verteilte Evolution von Produktlinien: Herausforderungen und ein Lösungsansatz, 1. Workshop Design For Future - Langlebige Softwaresysteme (L2S2), FZI Karlsruhe, 2009.

[SJ04] Schmid, K.; John, J.; A Customizable Approach to Full Lifecycle Variability Management; Sci-ence of Computer Programming, Vol. 53, No. 3, pp. 259-284, 2004.

[SLBWC11] She, S.; Lotufo, R.; Berger, T.; Wasowski, A.; Czarnecki, K.; Reverse engineering feature models. International Conference on Software Engineering, 461-470, 2011.

Die Objektorientierte Hülle – Erweiterbarkeit imperativ-prozeduraler Altsysteme durch Verschalung

Henning Schwentner, Jens Barthel

C1 WPS GmbH, Vogt-Kölln-Straße 30, 22527 Hamburg
{henning.schwentner, jens.barthel}@c1-wps.de

Für einen Kunden aus dem Leasinggeschäft war das Vertragsverwaltungssystem ERIKA gebaut worden. Die technische Basis ist SAP mit der Programmiersprache ABAP. Der ursprüngliche Entwurf war imperativ-prozedural. Als wir in das Projekt kamen, entstand der Wunsch, auch „moderne“ Techniken wie objektorientierte Programmierung zu verwenden. Damit sollte insbesondere die Testbarkeit verbessert werden.

Das System ist zwar imperativ entworfen, aber nicht ohne Architekturvorstellung. Insbesondere gilt die Architekturregel: „kein direkter Zugriff auf die Datenbank“. Stattdessen dürfen nur sogenannte Read- und Write-Funktionsbausteine aufgerufen werden, die den eigentlichen Datenbankzugriff kapseln. Auf diese Weise ruft die Geschäftslogik zwar die Datenbank nicht direkt, sie ist aber immer noch indirekt von der Datenbank abhängig.

Dies verursacht verschiedene Probleme. So ist die Geschäftslogik schwierig zu testen. Angenommen der Funktionsbaustein `BERECHNE_MARGE` ruft den Read-Funktionsbaustein `LIES_VERTRAG` auf. Ein Modultest, der `BERECHNE_MARGE` aufruft, ist dann automatisch auch von `LIES_VERTRAG` abhängig.

Wie löst man diese Abhängigkeit? Die von uns verwendete Variante ist die Einführung von „Objektorientierten Hüllen“. Dazu wird zu einem Read-Funktionsbaustein ein Interface erzeugt. Dieses Interface erhält nur eine Methode. Die Parameterliste dieser Methode entspricht der des verhüllten Funktionsbausteins. Im Beispiel zum Legacy-Funktionsbaustein `LIES_VERTRAG(importing VERTRAGSNUMMER VNR returning VERTRAG VT)` also ein Interface `IF_VERTRAGS_LESER` mit der Methode `LIES(importing VERTRAGSNUMMER VNR returning VERTRAG VT).`

Das Interface bekommt eine Standardimplementierung. In dieser wird die Methode so implementiert, dass sie den verhüllten Funktionsbaustein aufruft und alle Parameter eins-zu-eins weitergibt. Im Beispiel also Klasse `CL_VERTRAGS_LESER`, bei der die Methode LIES so implementiert wird, dass sie alle Parameter an `LIES_VERTRAG` weiterleitet.

Nun wird neue Geschäftslogik so geschrieben, dass sie statt dem Funktionsbaustein `LIES_VERTRAG` direkt nun die Methode `IF_VERTRAGS_LESER->LIES` des neuen Interface aufruft. Im Produktivbetrieb wird als Implementierung von `IF_VERTRAGS_LESER` die Klasse `CL_VERTRAGS_LESER` verwendet. Modultests schieben dem *object under test* als Implementierung von `IF_VERTRAGS_LESER` eine eigene Klasse unter, mit der die für den Test gewünschten Rückgabewerte injiziert werden können.

In dem Legacy-System wurden fachlich zusammenhängende Datensätze ausschließlich durch Fremdschlüsselbeziehungen in der Datenbank repräsentiert. Um diese Beziehungen zu verdeutlichen und einen einfacheren Zugriff auf einzelne Komponenten fachlicher Entitäten zu gewährleisten, bot es sich an, diese als Klassen nachzubilden.

So gab es z. B. eine Tabelle VTKOPF für die Vertragskopfdaten und eine Tabelle VTPOSI für Vertragspositionsdaten. Es bot sich an, für die fachliche Entität „Vertrag“ eine Klasse CL_VERTRAG einzuführen. Die Implementierung dieser Klasse arbeitet dann auf einer Struktur vom Typ VTKOPF. Wenn die Struktur Felder wir VTNR (für Vertragsnummer) oder GBEREI (für Geschäftsbereich) anbot, so führten wir Getter- und Setter-Methoden wie GIB_VERTRAGSNUMMER oder SETZE_GESCHAEFTSBEREICH ein. So erhielten wir etwas ähnliches wie eine JavaBean.

Im nächsten Schritt führten wir in die Klasse CL_VERTRAG Vor- und Nachbedingungen ein. Zum Beispiel zur Überwachung von Statusübergängen. Dann fügten wir „richtige“ Geschäftslogik hinzu. CL_VERTRAG erhielt dann z.B. eine Methode ABRECHNEN.

In ABAP gibt es sogenannte klassische Ausnahmen und objektorientierte Ausnahmen. Die klassischen Ausnahmen sind einfach benannte Returncodes, die nicht weitergeworfen werden. Dies bedeutet, dass die Fehlerbehandlung immer direkt nach dem Aufruf einer Funktion mit klassischen Ausnahmen gemacht werden muss.

In ERIKA sind die Read- und Write-Funktionsbausteine mit klassischen Ausnahmen versehen. Im Rahmen der Einführung von objektorientierten Hüllen haben wir die verhüllenden Methoden gleich mit objektorientierten Ausnahmen versehen. Die verhüllenden Methoden prüfen den Returncode der verhüllten Funktion und werfen bei einem Fehler eine passende objektorientierte Ausnahme.

Neue Entwicklungen konnten nun so entworfen werden, dass sie einerseits die objektorientierten Hüllen von bestehendem Coding statt des Codings selbst aufrufen konnten. Das hatte wieder den Vorteil, dass dieses neue Modul einfach mit einem Modultest versehen werden konnte. Es war sogar möglich, hier testgetrieben vorzugehen, d. h. dass wir durch die neue Infrastruktur erst ein Stück Test, dann ein Stück Code usw. schreiben konnten. Außerdem musste das neue Modul nicht auf Strukturen und Datenbanktabellen wie VTKOPF aufsetzen. Stattdessen konnten wir hier die höhere Abstraktion CL_VERTRAG verwenden.

Durch Einführung von objektorientierten Hüllen und Fachlichen Metaphern (Materialien) als Klassen ermöglichen wir, ein Altsystem auch mit modernen Mitteln weiterzuentwickeln. Dies ist genau dann gut möglich, wenn das Altsystem über einen zentralisierten Datenbankzugriff verfügt. Dieser kann dann relativ einfach verkapselt werden. Ohne zentralisierten Zugriff müsste jeder einzelne Datenbankzugriff ersetzt werden.

Für neue Module ist ein objektorientierter Entwurf und eine objektorientierte Implementierung möglich. Die neuen Module können leicht mit Modultests versehen werden, was u. a. Test-First-Programming ermöglicht. Dabei ist auch die Verwendung von Techniken wie Dependency Injection und Mocking möglich.

Das hier beschriebene Vorgehen wurde mit ABAP umgesetzt, es ist allerdings nicht auf diese Programmiersprache begrenzt und sollte genauso in anderen Umgebungen funktionieren, in denen imperative Systemteile nun auch objektorientiert verwendet werden können sollen.

ENVISION2020 – 3. Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme

An Artifact-oriented Framework for the Seamless Development of Embedded Systems

Wolfgang Böhm, Andreas Vogelsang
Technische Universität München
Institut für Informatik
Boltzmannstr. 3
85748 Garching b. München
{boehmw,vogelsan}@in.tum.de

Abstract: Transferring novel modeling concepts and approaches into a well established and customized industrial context is not easy. They have to be mapped to the specific development process of the application domain, must complement the existing tools, and exhibit certain representations. Artifact-oriented development distinguishes between the development process and the created artifacts in the context of a given development project. This paper provides a conceptual framework that encompasses an artifact-oriented view onto the development of embedded systems. We argue that this artifact-oriented view provides means to map academic models and description techniques onto existing development processes in industry. It furthermore provides the basis for the definition of tracing links and dependencies between the different contents and artifacts, allowing for a seamless development of artifacts.

1 Artifact-oriented Development

Over the years, a number of methods, processes, description techniques, and models have been proposed by academia in order to enhance the development of embedded systems. On the other side, there exists a plethora of well-established tools, development processes and best practices applied in industry. Therefore, the transfer of new ideas and approaches into a well-established and customized industrial context is not easy. Artifact-oriented development distinguishes between the development process, which might be very specific, and the created artifacts in the context of a given development project. It specifically aims at a detailed description of the structure, the content and the used concepts of the artifacts. We argue that this artifact-oriented view provides means to map academic models and description techniques onto existing development processes in industry. Additionally, it has a positive impact on the syntactic and semantic quality of the created artifacts [Me11].

An **artifact** is seen as a structured abstraction of modeling elements used as input, output, or as an intermediate result of the development process [Me10]. Artifacts capture and document information about the system, its development, and its context. Thus, they document system properties. This leads to an artifact model, which contains the artifacts to be developed during a specific development process together with their internal structure and dependencies between them.

In **artifact-oriented development**, the entire development process is understood as the stepwise construction of artifacts, always focusing on the results (the artifacts) that need to be produced during a development rather than focusing on the methods and processes that create them. We speak of "artifact-driven" versus "process-driven" development.

The basic assumption of artifact-oriented development is that, although development processes vary heavily from project to project, the content within the different artifacts is described by modeling concepts that are independent from the underlying development process. Note that content here refers to the logical content of an artifact, abstracting from its actual representation.

As artifact-oriented development puts emphasis on consistent result structures and used terminology, a given artifact can be created using quite different methods, processes and representations. The underlying development process is then just an arrangement of the artifacts produced during system development, together with the methods that produce them. This leads to a flat method structure. On the opposite, an activity-based (process driven) approach puts emphasis on how to produce something (rather than what to be produced) with a more vague description of content and structure, producing a flat artifact structure, where dependencies between artifacts only arise from dependencies between the methods that create them. It should be noted that even in a process-centric environment, such as the development of automation software, an artifact-oriented approach can be applied by filtering the results of the various process steps and abstracting from the methods to produce them in the first place.

Artifact orientation comes with various interpretations and manifestations in practice. Therefore, we need a clear definition of the term artifact itself. There is a variety of information that is embodied in an artifact. We distinguish between:

- the structure of the artifact (e.g., given by a table of content)
- the artifacts logical content, i.e., the pure assertions about a system
- the modeling concepts, i.e., the language by which the logical content is expressed
- the representation (including description techniques) of the artifacts content (e.g., natural text, diagrams, models, tables)
- the dependencies between the logical content

Contribution: This paper provides a conceptual framework that encompasses an artifact-oriented view onto the development of embedded systems by introducing two different models: The artifact model and the concept model. In a nutshell, the concept model defines modeling concepts that are used to describe a set of content items (e.g., elements and relations necessary to specify a state machine). These content items are arranged in a process-dependent hierarchical artifact structure that is defined in the artifact model. The artifact model contains the set of artifacts that need to be produced within the specific engineering process together with the dependencies and relations between them. Each artifact has a hierarchical structure (a table of content) of (sub-) artifacts with leafs being the

various content items. Each content item of an artifact is linked to a concept of the concept model by a specific representation of the concept.

Upon such a framework, we are able to provide concepts for different content items independent from the engineering process. On top of that, we can define clear responsibilities and support a progress control for the production of artifacts. This framework can also be used to compare different engineering processes (e.g., from different application domains) and to pinpoint potential needs for optimization. Furthermore, the framework provides the basis for the definition of tracing links and dependencies between the different contents items and artifacts, allowing for a seamless development.

After introducing this conceptual framework, we instantiate it by providing a concept model for the development of embedded systems as it is worked out in the SPES_XT[1] project. We exemplarily show how this concept model can be linked to a given engineering process in industry by providing a process-dependent artifact model. We furthermore show the benefits of this approach by highlighting dependencies between artifacts, which need to be maintained in that engineering process.

2 Related Work

Artifact orientation has gained much attention in recent years, especially in requirements engineering approaches. In these approaches, artifact orientation is used to define RE reference processes and connect them with special concepts that are used within these processes.

REMsES [Br10] provides a process guide for supporting requirements engineering processes in the automotive industry. This approach is based on three models: an artifact model, a process model, and an environment model. The artifact model provides a basic structure for the definition of the artifacts, their assignment to abstraction layers and content categories, and the relations between the artifacts. It defines general control flow dependencies within requirements engineering processes. The process model defines the coarse-grained course of action and fine-grained task descriptions. It defines individual artifact-related tasks. The environment model defines the interfaces between the environmental processes that interact during the engineering process of the system with its requirements engineering process. The approach in this paper follows this idea and extends it to be applicable for the entire engineering process and not just RE. Additionally, it also focuses on the concepts used to define parts of the artifacts allowing for a precise definition of dependencies between artifacts.

REMbIS [Me10] is a model-based RE approach for the application domain of business information systems. It consists of (1) an artifact abstraction model that defines horizontal abstraction and modeling views, (2) a concept model that defines those aspects dealt with during construction of models including the definition of possible notions for producing the models and finally (3) a generic process model with milestones, phases, and roles that

[1] http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html

defines the activities and tasks of the RE process. We follow this approach in large parts and extend it to capture the entire development process.

3 Conceptual Framework for an Artifact-oriented Development

The conceptual framework introduced in this section distinguishes between the process-dependent set and structure of artifacts, defined in an artifact model, and the process-independent use of concepts used to describe certain content, defined in the concept model.

The concept model is a collection of modeling concepts together with dependencies that may exist between the different concepts. Each concept is characterized by an ontological basis, which describes the ontological entities of the concepts and the relations between them (see Figure 1). Thus, a concept defines an abstract syntax of a modeling language. Besides this pure description of the syntax, the concept model could also provide semantics for the concepts used. A concept model captures the complete vocabulary of the engineering tasks necessary to develop a system. Therefore, it provides modeling languages for a well-defined, structured specification of the content, while at the same time abstracting from the actual representation used in a specific artifact (e.g. tables, plain text, models, code). Concept models can have different levels of "richness" with regard to how expressive and customized the defined concepts are. Simple concept models could just provide general modeling concepts like state machines or Petri nets. Richer concept models could provide more specialized concepts, which build upon such simple concepts in order to define a more specific content. A specialized concept for the definition of a system function could use the concept of a state machine to describe a functions behavior.

Since systems and their descriptions and documentation can get very large, it is essential to adequately structure the produced artifacts as well as their logical content. Therefore, we define an artifact model in the conceptual framework that defines the artifacts produced in a development process as a hierarchical structure with specific content items as leafs (see Figure 1). The artifact model defines for each artifact the expected content (e.g., defined by means of a taxonomy). In this model, content items serve as containers for the actual content and define single areas of responsibility. In addition, we can regard these content items as artifacts by themselves and different content items may be grouped together to form another artifact at a higher level, which may be the outcome of a specific process step. This leads to a hierarchical structure of artifacts with content items being the leaves of the tree (see Figure 2).

As an example, the reader may consider a system specification as a concrete artifact to be produced during the development process. The structure of this specification (e.g., given by its table of content) will also be described in the artifact model. The concepts used to describe the individual pieces of content within the system specification are described in the concept model. To link the two models together, the concepts are related to content items. In doing so, a concept is given a specific representation that is used to describe the concept within an artifact. The concept now becomes content that appears in one or more artifacts, depending on the development process being used. However, the representation

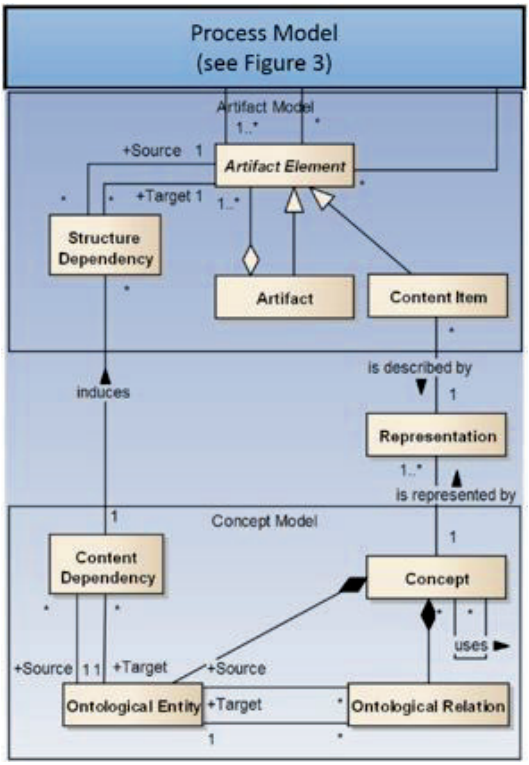


Figure 1: Meta Model of the Conceptual Framework (cf. [Me10])

of the concept might vary depending on the artifact in which it is expressed. Therefore, we say that a concept has a number of representations (e.g., diagram, text, or table), which are used to describe different content items in artifacts. As a simple example of the above, we consider the concept of a state machine, which can be used to describe the behavior of a logical component in the architecture of a system. The development process used may call for a system architecture specification, which includes the state machines of all components as state transition diagrams. Another way to describe these state machines is a tabular representation, which could be used in an interface specification document. The representation link also allows for the use of different concrete modeling languages. For example in an avionic context, a state machine might be represented by a SysML State Machine Diagram, whereas in an automotive context this might be expressed using a Simulink Stateflow Chart.

Please note that the concept model does not only cover content items that appear in documents. It also includes content that arises from producing code for example. In this case the appropriate concept might be a specific programming language or a structural element (e.g., a class or a method).

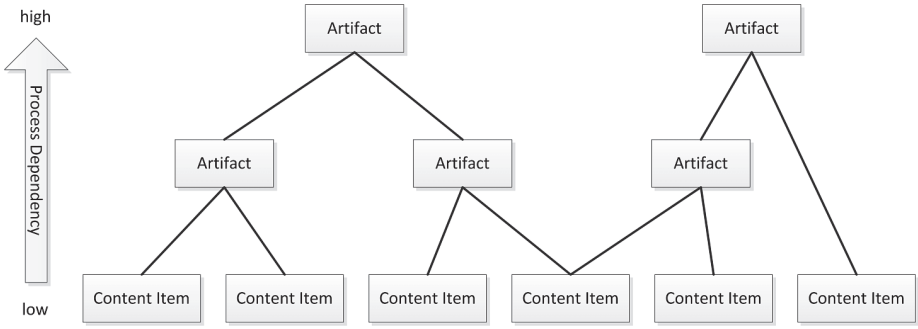


Figure 2: Artifact Hierarchy

A special challenge in a structured system development approach is an appropriate handling of dependencies. In our framework we aim at a precise description of dependencies between concepts, content items, and artifacts. Dependencies on the level of concepts describe the relation between ontological entities of different concepts. These inter-concept dependencies express dependencies with regard to content and are independent of the development process (e.g., events of a state machine must be consistent with the interface of the logical component in which they are embedded).

When concepts are mapped to the artifact model and thus become content items in a specific artifact, these content dependencies induce dependencies between content items and therefore also between artifacts. The induced dependencies arise from the chosen artifact structure and constrain the way the artifacts can be created. Please note that the dependencies between the content items are not limited to a single artifact. An artifact model thus defines a description of the set of required artifacts, their structure and contents, and the relations between the artifacts.

4 Mapping Content and Artifacts to a Development Process

The content items of the artifacts together with the related concept model and their dependencies can be regarded as a blue-print of a comprehensive system specification covering the whole development process. Therefore, an artifact model can be used as a reference model that captures the domain-specific results of the development steps. As in artifact-oriented development the content items are independent of the development process there must be a mapping of the artifact model to the actual development process.

This mapping is established by assigning the artifacts of the artifact model to tasks and milestones of the development process. Conversely, we can obtain content items from a given development process by filtering the results of the various process steps and abstracting from the methods to produce them.

Figure 3 provides a meta model for this mapping. The generic process model structures a development process into a set of milestones and tasks. Each artifact is assigned to a milestone where it has to be delivered. Artifacts are produced within tasks, in which potentially other artifacts are needed as inputs in order produce new artifacts.

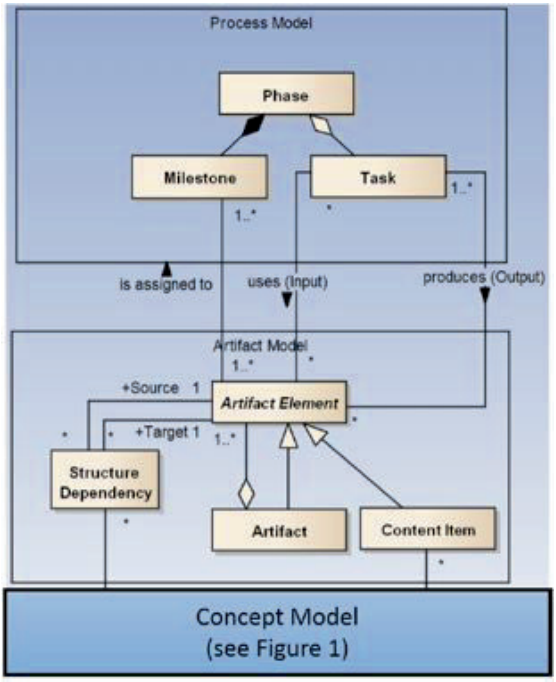


Figure 3: Mapping of the Artifact Model onto a Generic Process Model (cf. [Me10])

5 Example: Mapping the SPES Modeling Framework to the V-Model XT Process

The SPES Modeling Framework, as described in [Br12], provides a structured set of model types that are considered beneficial for the development of embedded systems. These model types are structured into so called Viewpoints, which group the model types according to some concerns following the standard of IEEE42010. Each model type has an ontological basis. Thus, the SPES Modeling Framework can be considered as a concept model. However, the model types in SPES do not only cover the information about the concepts to be used. They additionally provide information about the content that should be addressed by using these concepts. Therefore, the model types in SPES can also be considered as a basic set of content items that can/need to be created in the development

of an embedded system. In summary, the SPES Modeling Framework defines a set of content items (basic artifacts) together with concepts that are used to describe the content items. In the following, we will map these content items and concepts to an artifact model instance for development process V-Model XT [Fr09].

The V-Model XT is a development process meta model that needs to be instantiated for a given project context. The instantiation provides an organisational tailoring of considered roles, activities, and products to be produced during the development. In the context of the V-Model XT, artifacts are called (work)products. For this paper, we exemplarily examine the high-level products "Anforderungen (Lastenheft)"[2] and "Gesamtsystemspezifikation (Pflichtenheft)"[3]. We will create an instance of an artifact model containing these two products (artifacts) and show how the content items of the SPES Modeling Framework can be related to these artifacts. Figure 4 shows the artifact model for the two products. The mapping of the content items to the artifacts is based on the textual descriptions given for the products and the SPES Modeling Framework. Note, that this model is not complete. Both products contain further content that is not considered here.The V-Modell XT documentation additionally provides a mapping for these two products to tasks in which they need to be created.

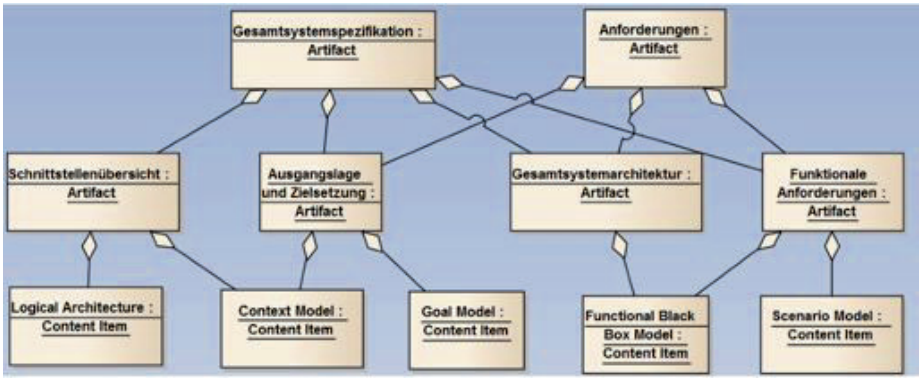


Figure 4: Artifact model instance for the V-Modell XT products "Gesamtsystemspezifikation" and "Anforderungen".

The SPES Modeling Framework also defines dependencies between the model types that are expressed on the level of the used concepts. There is for example a dependency defined which states that the interface behavior of the Functional Black Box Model must be refined by the interface behavior of the Logical Architecture. This dependency on the level of concepts induces a dependency between the content items and finally between the artifacts

[2] http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.4/Dokumentation/V-Modell%20XT%20HTML/14794f684e963e8.html#ref14794f684e963e8

[3] http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.4/Dokumentation/V-Modell%20XT%20HTML/f436f8cfc083ae.html#reff436f8cfc083ae

to which they are linked to in the artifact model. Exactly this dependency between the two artifacts is also described in the V-Modell XT documentation[4]. However, with the definition of dependencies on the level of concepts we can be much more precise and process-independent.

6 Conclusion and Outlook

In this paper, we have shown that an artifact-oriented view can be used to structure the system development into process-dependent and process-independent parts. This separation is useful in order to assess and classify academic development approaches and to map them to specific development processes used in industry. This does not only cover specific tasks and artifacts used in the process but also specific representations used.

The presented conceptual framework is open to increments in order to cover additional aspects. One aspect, for example, could cover the artifacts life cycle by extending the artifact model with attributes that entail the current state of an artifact (e.g., draft, in review, released).

A further benefit of the presented framework is that it opens the way to a fully integrated tooling environment, in which content items, which are described by concepts can be linked to specific tools that are used in order to create these content items. The dependencies in the concept model clearly state the connection between the models within tools that need to be maintained during development. For practical purposes the content items can be stored in a content repository, similar to a Product-Lifecycle-Management system (PLM). The relations between the content items are maintained by the repository system such that changes in one content item are automatically updated in all related content items. Given the artifact model, the actual artifact documents can be generated by compiling the corresponding content items from the repository. This way, artifacts always reflect the current state of development and have a higher quality.

References

[Br10] Peter Braun, Manfred Broy, Frank Houdek, Matthias Kirchmayr, Mark Müller, Birgit Penzenstadler, Klaus Pohl, and Thorsten Weyer. Guiding requirements engineering for software-intensive embedded systems in the automotive industry. *Computer Science - Research and Development*, 2010.

[Br12] Manfred Broy, Werner Damm, Stefan Henkler, Klaus Pohl, Andreas Vogelsang, and Thorsten Weyer. Introduction to the SPES Modeling Framework. In Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy, editors, *Model-Based Engineering of Embedded Systems*. Springer Berlin Heidelberg, 2012.

[4] `http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.4/Dokumentation/V-Modell%20XT%20HTML/18296108af1c73c3.html#ref18296108af1c73c3`

[Fr09] Jan Friedrich, Ulrike Hammerschall, Marco Kuhrmann, and Marc Sihling. Das V-Modell XT. In *Das V-Modell XT*, Informatik im Fokus. Springer Berlin Heidelberg, 2009.

[Me11] Daniel Méndez Fernández, Klaus Lochmann, Birgit Penzenstadler, and Stefan Wagner. A case study on the application of an artefact-based requirements engineering approach. In *Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, 2011.

[Me10] Daniel Méndez Fernández, Birgit Penzenstadler, Marco Kuhrmann, and Manfred Broy. A Meta Model for Artefact-Orientation: Fundamentals and Lessons Learned in Requirements Engineering. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010.

Ein strukturierter Ansatz zur Ableitung methodenspezifischer UML/SysML-Profile am Beispiel des SPES 2020 Requirements Viewpoints[1]

Bastian Tenbergen, Philipp Bohn, Thorsten Weyer

paluno – The Ruhr Institute for Software Technology
Universität Duisburg-Essen
Gerlingstraße 16, 45127 Essen
{bastian.tenbergen | philipp.bohn | thorsten.weyer}@paluno.uni-due.de

Abstract: Eine wesentliche Voraussetzung für die industrielle Akzeptanz von wissenschaftlich entwickelten modellbasierten Entwicklungsmethoden ist, dass sich die Methode in die Werkzeug- und Prozesslandschaft von Industrieunternehmen eingliedert und somit angewendet werden kann, ohne dass methodenspezifische Werkzeuge notwendig sind. Profile erlauben es, UML/SysML für spezielle Entwicklungsmethoden anzupassen, indem Konzepte der Methode in UML/SysML abgebildet werden. Eine Herausforderung bei der Entwicklung von UML/SysML-Profilen besteht u.a. darin, die Profile systematisch und strukturiert abzuleiten, sodass die Konzepte der Methode auf UML/SysML korrekt und vollständig abgebildet werden können. Während in der Literatur Ansätze zur Ableitung domänenspezifischer UML/SysML-Profile existieren, gibt es bisher jedoch keinen Ansatz zur Ableitung methodenspezifischer Profile. Ziel dieses Artikels ist es daher, einen strukturierten Ansatz zur Definition solcher Profile vorzustellen. Zunächst wird dazu die wesentliche Literatur zur strukturierten Definition von domänenspezifischen Profilen untersucht. Anschließend wird ein auf bestehender Literatur basierender Ansatz vorgestellt und anhand des SPES 2020 Requirements Viewpoints illustriert. Durch diesen Ansatz wird eine Grundlage für die Anwendbarkeit einer modellbasierten Entwicklungsmethode insofern geschaffen, als dass die Konzepte der Methode auf UML/SysML abgebildet werden können und die Entwicklungsmethode mit im spezifischen Industriekontext bereits vorhandenen UML-Modellierungswerkzeugen angewendet werden kann.

1 Einführung und Motivation

Eine Möglichkeit, die Herausforderungen bei der Entwicklung heutiger softwareintensiver eingebetteter Systeme zu adressieren, ist die Verwendung von modellbasierten Entwicklungsansätzen, wie z.B. [Br12]. Allerdings haben verschiedene Studien zum Stand der Praxis modellbasierter Entwicklung (z.B. [LPR93]) gezeigt, dass modellbasierte Entwicklungsansätze nur zögerlich in der Industrie eingeführt werden, u.a. auf Grund von fehlenden Nutzensnachweisen im industriellen Kontext [STP12]. Eine wesentliche Voraussetzung für solche Industriestudien ist jedoch, dass die Ansätze im in-

[1] Dieser Beitrag wurde im Rahmen der BMBF-Projekte SPES 2020 (Förderkennzeichen: 01IS08045V) und SPES 2020_XTCore (Förderkennzeichen: 01IS12005C) gefördert.

dustriellen Kontext anwendbar sind, u.a. durch flexible Integration der Entwicklungsansätze in die bestehenden Werkzeugketten [GLT03], Wiederverwendung bestehender Werkzeuge sowie durch den Einsatz gängiger Modellierungssprachen [Da06], wie beispielsweise UML und SysML.

Durch den Einsatz von UML/SysML-Profilen kann diese Voraussetzung geschaffen werden [FV04]. Da Profile konzeptueller Natur und somit werkzeugunspezifisch sind, können sie in der Regel für spezifische Werkzeuge implementiert und in bestehende Werkzeug- und Prozessketten eingegliedert werden. In Abbildung 1 wird dieser Sachverhalt dargestellt: ein konzeptuelles Profil wird für zwei unterschiedliche Werkzeuge spezifisch angepasst. Somit passen sich die werkzeugspezifischen Implementierungen in die bestehende Werkzeugkette ein und entsprechen gleichzeitig dem konzeptuellen Profil. Da Profile konzeptueller Natur sind, entfällt somit die Notwendigkeit Werkzeug- und Prozessketten durch Einbeziehen von neuen methodenspezifischen Modellierungswerkzeugen anzupassen und Entwickler im Umgang mit zusätzlichen Werkzeugen zu schulen.

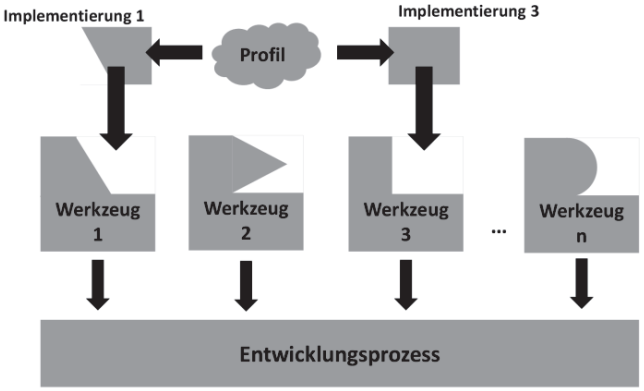


Abbildung 1: Werkzeugunterstützung durch UML/SysML-Profile

Profile werden typischerweise zur Abbildung von Konzepten spezifischer Anwendungsdomänen, wie z.B. Automatisierungstechnik auf das UML-Metamodell verwendet. Durch Profile kann UML/SysML erweitert werden, indem fehlende Konzepte, d.h. Typen oder Klassen mit besonderen semantischen Eigenschaften, die über die der UML eigenen Typen hinausgehen, hinzugefügt werden. Zur Erstellung von UML/SysML-Profile bedarf es jedoch systematischen Vorgehensweisen, da andernfalls die Gefahr besteht, dass im Hinblick auf den werkzeugtechnisch zu unterstützenden Entwicklungsansatz unvollständige oder inkorrekte Profile gebildet werden und somit u.a. die Integration des Entwicklungsansatzes in bestehende Werkzeuge beeinträchtigt wird.

Während die Literatur einige Ansätze zur Ableitung domänenspezifischer Profile vorschlägt, existiert allerdings kein Ansatz zur systematischen Ableitung methodenspezifischer Profile. In diesem Beitrag wird solch ein Ansatz vorgestellt.

In Abschnitt 2 werden zunächst der Stand der Wissenschaft zur Erstellung von UML/SysML-Profilen zusammenfassend diskutiert und die notwendigen Qualitätseigenschaften für methodenspezifische Profile expliziert. In Abschnitt 3 wird anschließend ein Ansatz vorgestellt, der es gestattet, auf systematische Art und Weise Profile für Modellierungsmethoden zu erstellen, die diese Eigenschaften aufweisen. Abschnitt 4 illustriert die Anwendung des Ansatzes anhand eines konkreten Beispiels.

2 UML/SysML-Profile: Grundlagen und Entwicklung

UML/SysML-Profile sind domänenspezifische Modellierungssprachen (DSML), die das UML-Metamodell [MOF06] im Hinblick auf die Bedürfnisse von speziellen Anwendungsdomänen erweitern ([MOF06], [KT08]), indem sie Stereotypen definieren. UML/SysML-Profile können beispielsweise domänenspezifische Konzepte der Automotive-Domäne (z.B. „Drehmoment“ und „Motor“) und die Beziehungen zwischen Konzepten (z.B. „Motor erzeugt Drehmoment“) dem UML-Metamodell hinzufügen, indem bereits existierende Sprachelemente konsistent zu den im UML-Metamodell definierten Regeln wiederverwendet werden [FV04]. In modellbasierten Entwicklungsansätzen können UML/SysML-Profile zudem eingesetzt werden, um spezifische Artefakttypen von Entwicklungsansätzen und Methoden abzubilden, indem UML/SysML-Diagrammarten wiederverwendet und spezifisch um ggf. fehlende Konzepte erweitert werden. Methodenspezifische UML/SysML-Profile erweitern folglich die UML/SysML-Syntax und schaffen darüber hinaus eine Grundlage für die semantische Integration einer Methode, indem sie die spezifischen Konsistenzregeln der Modellierungsmethode berücksichtigen müssen und somit ggf. auch Einschränkungen bei der Verwendung der UML/SysML-Notation nach sich ziehen.

2.1 Ansätze zur systematischen Entwicklung von UML/SysML-Profilen

Bezüglich der Literatur zur Erstellung von domänenspezifischen Sprachen und UML/SysML-Profilen kann u.a. zwischen „leichtgewichtigen“ und „schwergewichtigen“ Ansätzen unterschieden werden [WS07]: leichtgewichtige Ansätze eignen sich für Situationen, in denen bestehende Konzepte des UML-Metamodell weitestgehend zur Abbildung der zu modellierenden Domäne ausreichen [La08]. Schwergewichtige Ansätze erlauben darüber hinaus, die Semantik der Modellierungssprache zum Zwecke spezifischer formaler Analysen anzupassen [AP08, FP10].

Lagarde et al. beschreiben in [La08] einen leichtgewichtigen Ansatz zur systematischen Erstellung von UML/SysML-Profilen. Bei der Anwendung des Ansatzes wird zunächst die zu modellierende Domäne analysiert, die wesentlichen im Profil zu berücksichtigen Konzepte identifiziert und als Stereotypen sowie Relationen zwischen den Stereotypen abgebildet. Anschließend wird ein Grundgerüst des Profils erstellt und mögliche Inkonsistenzen identifiziert und aufgelöst. Der Ansatz kann somit die Vollständigkeit des Profils sicherstellen, d.h. alle für den Modellierungszweck notwendigen Domänenkonzepte im Profil berücksichtigen. Ein weiterer leichtgewichtiger Ansatz wird von Selic in [Se07] vorgestellt. Ähnlich dem leichtgewichtigen Ansatz nach Lagarde et al. analysiert

der Ansatz nach Selic ebenfalls die zu modellierende Domäne. Anschließend werden die identifizierten Konzepte jedoch direkt in Stereotypen übertragen und keine besondere Qualitätssicherung durchgeführt.

Ein schwergewichtiger Ansatz wird von Kelly und Tolvanen in [KT08][2] vorgeschlagen. Der Ansatz von Kelly und Tolvanen basiert ebenfalls auf einer Analyse der zu modellierenden Domäne und identifiziert für die Modellierung wesentliche Konzepte. Im Gegensatz zu den leichtgewichtigen Ansätzen wird danach ein domänenspezifisches Metamodell formalisiert und eine Semantik definiert, indem ein Bezug zwischen identifizierten Domänenkonzepten und abgebildeten Metamodellelementen formal hergestellt wird. Anschließend können zusätzliche Stereotypen definiert und gegen das zuvor erstellte Domänenmodell validiert werden.

2.2 Anforderungen an einen Ansatz zur systematischen Entwicklung von methodenspezifischen UML/SysML-Profilen

Bei den vorgestellten Ansätzen zur Entwicklung von UML/SysML-Profilen lässt sich ein grundlegendes Muster erkennen: Zunächst wird die Domäne betrachtet und die wesentlichen Konzepte identifiziert. Diese werden im UML/SysML-Profil durch neue Stereotypen repräsentiert und anschließend validiert. Durch die Validierung wird sichergestellt, dass das resultierende Profil in Bezug auf die zu modellierende Domäne vollständig und korrekt und auch konsistent zu den bereits existierenden Konzepten in UML/SysML ist. Ein Ansatz zur systematischen Entwicklung methodenspezifischer UML/SysML-Profile muss demnach Profile mit den folgenden Qualitätseigenschaften erstellen können:

- *Vollständigkeit*: Der Ansatz muss Profile entwickeln, die in Bezug auf die zu unterstützende Entwicklungsmethode insofern vollständig sind, als dass alle Artefakte der Entwicklungsmethode im Profil repräsentiert werden.
- *Korrektheit*: Der Ansatz muss Profile entwickeln, die in Bezug auf die zu unterstützende Entwicklungsmethode insofern korrekt sind, als dass Konsistenzregeln zwischen Artefakten der Entwicklungsmethode nicht verletzt werden.
- *Konsistenz*: Der Ansatz muss Profile entwickeln, die in Bezug auf die zu unterstützende Entwicklungsmethode insofern konsistent sind, als dass das Profil nicht syntaktisch und semantisch inkompatibel mit dem UML-Metamodell ist und im größtmöglichen Umfang bestehende UML/SysML-Konzepte wiederverwendet.

3 Strukturierter Ansatz zur Entwicklung von UML/SysML-Profilen

Für die Entwicklung eines werkzeugunabhängigen, strukturierten Ansatzes für konzeptuelle Profile wurde der leichtgewichtige Ansatz nach Lagarde et al. (siehe Abschnitt 2.1) adaptiert. Die spezifischen Arbeitsschritte des Ansatzes wurden in dem Umfang

[2] Der Ansatz befasst sich mit der Neuentwicklung domänenspezifischer Sprache, kann aber auf UML/SysML-Profile angewendet werden.

angepasst, wie es für die Erstellung methodenspezifischer Profile notwendig ist. Der Ansatz setzt sich aus den folgenden Schritten zusammen:

- *Schritt 1*: In diesem Schritt werden die wesentlichen Artefakttypen der Entwicklungsmethode identifiziert und in eine initiale Ontologie übertragen. Dies entspricht dem Erstellen des Domänenmodells im Ansatz nach Lagarde et al., unterscheidet sich von Schritt 1 in [La08] jedoch insofern, als dass die von der Methode verwendeten Modelltypen mit den darin enthaltenen Artefakten (z.B. „Zielmodell“, siehe Abschnitt 4.1) dokumentiert werden.
- *Schritt 2*: In diesem Schritt wird die initiale Ontologie validiert und ggf. fehlende Konzepte der Methode ergänzt. Außerdem werden in diesem Schritt die Abhängigkeiten und Konsistenzregeln zwischen den Modell- und Artefakttypen der Methode (z.B. „Ein Szenario erfüllt mindestens ein Ziel“, siehe [Da12], S. 64) ebenfalls in der Ontologie dokumentiert. Dadurch wird sichergestellt, dass das resultierende Profil korrekt und konsistent ist, d.h. nicht der Methode bzw. dem UML-Metamodell widerspricht. Dieser Schritt entspricht dem dritten Schritt in [La08], berücksichtigt aber zusätzlich zu den syntaktischen Vorgaben des UML-Metamodells noch die semantischen Beziehungen der Artefakttypen innerhalb der Modellierungsmethode.
- *Schritt 3*: In diesem Schritt können die gefundenen Konzepte in ein Konzeptmodell der technischen Umsetzung überführt und auf existierende Konzepte des UML-Metamodells abgebildet werden. Den methodenspezifischen Artefakttypen können äquivalente Modelltypen aus UML/SysML in diesem Schritt im Rahmen einer pragmatischen Implementierung (vgl. Abschnitt 1.5.3 in [BB12]) zugeordnet und somit im methodenspezifischen Profil wiederverwendet werden. Dieser Schritt entspricht dem Erstellen einer initialen Profilstruktur aus [La08], sieht jedoch im Besonderen die Wiederverwendung existierender Diagrammarten vor.
- *Schritt 4*: In diesem Schritt wird die initiale Profilstruktur aus Schritt 3 durch die noch fehlenden Stereotypen ergänzt, was Schritt 4 aus [La08] entspricht.

4 Anwendung des Ansatzes

In diesem Abschnitt wird anhand eines konkreten Beispiels die Umsetzung des Ansatzes aus Abschnitt 3 veranschaulicht. Als konkretes Beispiel wurde der SPES 2020 Requirements Viewpoint (ReqVP, [Da12]) gewählt.

4.1 Der SPES 2020 Requirements Viewpoint

Der SPES 2020 Requirements Viewpoint [Da12] zielt darauf ab, die Anforderungen an ein software-intensives eingebettetes System systematisch zu erheben und zu dokumentieren. Der ReqVP unterstützt den Requirements Engineer dabei, die Anforderungen so zu dokumentieren, dass zwischen lösungsneutraler Problembeschreibung und lösungskonzeptbezogenen Anforderungen unterschieden werden kann. Dazu besitzt der ReqVP vier wesentliche Modelltypen:

- *Kontextmodelle:* Dieser Modelltyp dokumentiert die Schnittstellen des geplanten Systems, Entitäten in der operativen Umgebung (z.B. andere Systeme oder Nutzer), die in Interaktion mit dem System stehen, sowie Ein- und Ausgaben, die zwischen System und Umgebung ausgetauscht werden.
- *Zielmodelle:* Dieser Modelltyp dokumentiert Ziele bzw. Intention der Stakeholder hinsichtlich der Systemfunktionalität sowie gewünschte Qualitäten des Systems. Zielmodelle dienen als Begründung für lösungskonzeptbezogene Anforderungen.
- *Szenariomodelle:* Dieser Modelltyp dokumentiert typische Interaktionen zwischen dem System und Entitäten der Systemumgebung und zeigt die beispielhafte Erfüllung konkreter Ziele.
- *Lösungskonzeptbezogene Anforderungsmodelle:* Dieser Modelltyp dokumentiert die funktionalen Anforderungen des geplanten Systems präzise und vollständig. Dabei wird zwischen statisch-strukturellen (z.B. die Informationsstruktur des Systems), operationalen (z.B. Anforderungen an konkrete Nutzerfunktionen) und Verhaltensanforderungen (z.B. von außerhalb des Systems erfahrbare Systemzustände) unterschieden und somit Anforderungen mit Bezug einem konkreten Lösungskonzept dokumentiert.

Anforderungen können im ReqVP auf unterschiedlichen Abstraktionsebenen der Systemdekomposition hinweg spezifiziert werden. Beginnend mit der Betrachtung des Gesamtsystems auf der höchstens Abstraktionsebene, wird mit jeder tieferen Systemeben das geplante System detaillierter beschrieben. Dabei werden auf jeder Abstraktionsebene alle im Requirements Viewpoint definierten Artefakttypen unterstützt.

Details zum Requirements Viewpoint der SPES 2020 Modellierungsrahmenwerkes, insbesondere zu den Konsistenzregeln können [Da12] entnommen werden.

4.2 Erstellung eines methodenspezifischen Profils für den ReqVP

In diesem Abschnitt wird beschrieben, wie der in Abschnitt 3 beschriebene Ansatz auf den SPES 2020 Requirements Viewpoint angewendet wird um ein konkretes SysML-Profil zu erstellen. Die Ergebnisse der einzelnen Arbeitsschritte werden in den Abbildungen dargestellt.

Schritt 1: Identifikation und Dokumentation der Methodenkonzepte
Abbildung 2 zeigt die initiale Ontologie des ReqVP als UML-Klassendiagramm. Die Ontologie spezifiziert die für die in Abschnitt 4.1 beschriebenen Modelltypen notwendigen Artefakttypen (z.B. *Goal*, *Refinement* und *Goal Dependency* für Zielmodelle) und setzt diese zueinander in Beziehung. Neben den für die Modelltypen wesentlichen Konzepten wurden außerdem artefaktübergreifende Konzepte definiert. So wurden beispielsweise die Konzepte *System*, *Interface* und *Stakeholder* definiert, da diese sowohl für Szenariomodelle als auch für Kontextmodelle relevant sind.

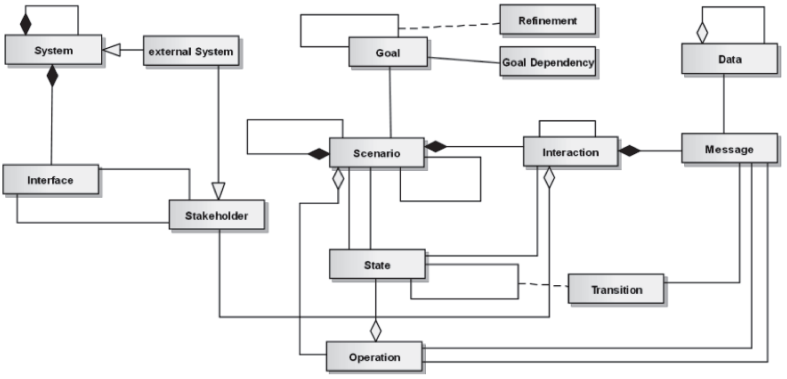


Abbildung 2: Die initiale Ontologie

Schritt 2: Qualitätssicherung und Vervollständigung des Konzeptmodells
Anschließend wurde die Ontologie validiert und vervollständigt. Beispielsweise wurden spezifischen Arten von *Goal*, *Refinement* und *Goal Dependency* definiert, da diese Spezialisierungen wesentliche Auswirkungen auf die Bedeutung des mit dem ReqVP modellierten Sachverhalts haben. Außerdem wurde das Konzept *Stakeholder* verfeinert und gegen einen abstrakten[3] *Actor* ausgetauscht und spezialisiert. Anschließend wurde gemäß der Konsistenzregeln des SPES 2020 Requirements Viewpoints (siehe [Da12], S. 64) Assoziationsnamen, Multiplizitäten, Rollen und Leserichtungen der Ontologie hinzugefügt. Das Ergebnis dieses Arbeitsschrittes ist in Abbildung 3 dargestellt.

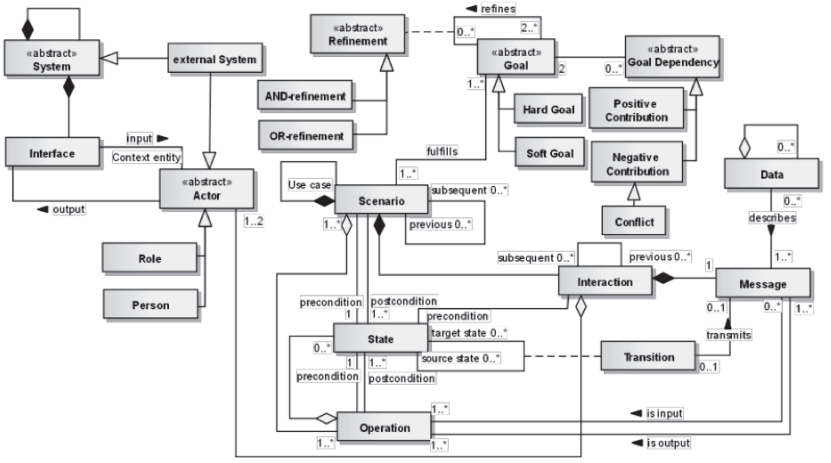


Abbildung 3: Die um Konsistenzmerkmale erweiterte Ontologie

[3] „Abstrakt“ bedeutet, dass nicht das ontologische Element selber, sondern eines seiner Spezialisierungen von einem konkreten Modellelement abgebildet wird.

Schritt 3: Erstellen eines Konzeptmodells der technischen Umsetzung
Im nachfolgenden Schritt wurde die vollständige methodenspezifische Ontologie aus Abbildung 3 in ein initiales Konzeptmodell für die technische Umsetzung des Profils überführt und konkreten UML/SysML-Diagrammarten zugeordnet. Da in diesem Beispiel ein SysML-Profil erstellt wird, wurde die Ontologie mit bereits bestehenden SysML-Konzepten verglichen, um ggf. äquivalente Diagramme aus SysML pragmatisch wiederzuverwenden („Pragmatische Implementierung“, [BB12]). Dabei hat sich beispielsweise gezeigt, dass Kontextmodelle und statisch-strukturelle Anforderungsmodelle durch SysML-Blockdiagramme dargestellt werden können. Es konnten also für diese Modelltypen die entsprechenden SysML-Diagrammarten wiederverwendet werden, was in Abbildung 4 durch die *<<import>>*-Beziehungen dargestellt wurde. Wie in Abbildung 4 entnommen werden kann, gibt es in SysML kein zu Zielmodellen äquivalentes Diagramm, sodass hierfür in Schritt 4 entsprechende Stereotypen abgeleitet werden müssen.

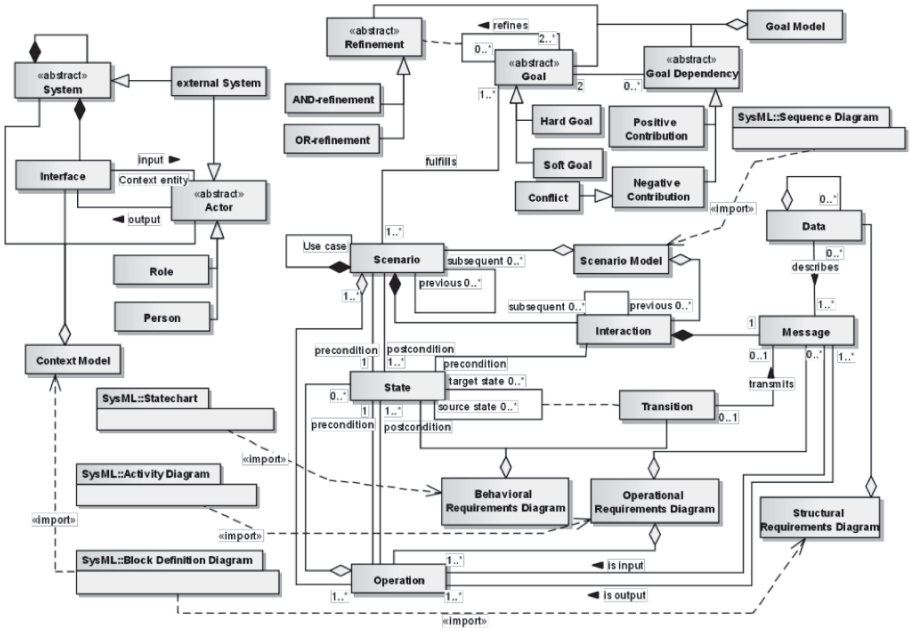


Abbildung 4: Konzeptmodell der technischen Umsetzung des Profils

Schritt 4: Definition fehlender Stereotypen
In diesem Schritt wurde für die zur Zielmodellierung notwendigen Artefakte Stereotypen definiert. Dazu wurde ein Auszug der im ReqVP verwendeten Konzepte des KAOS Rahmenwerks [Da12, La09] modelliert (vgl. Abbildung 5). Die Stereotypen *Goal* und *Refinement* erweitern die SysML-Metaklasse *Block*. *Refinement* realisiert die Konzepte *AND-* und *OR-Refinement* aus Abbildung 3. Da beide Konzepte durch das gleiche Notationselement modelliert werden und eine Unterscheidung nur auf Modellebene getroffen werden kann (vgl. [La09]), ist ein Stereotyp für beide Arten von *Refinement* ausreichend.

Die Konzepte *Refined By*, *Refinement Of* und *Contribution* stellen syntaktische und hinsichtlich ihrer Notation verschiedene Arten von Assoziationen zwischen Zielen dar, weshalb diese die Metaklasse *Association* bzw. *Generalization* erweitern. Spezielle Typen von *Goal* bzw. *Contribution* werden durch die Enumerationen *GoalType* und *ContributionType* festgelegt.

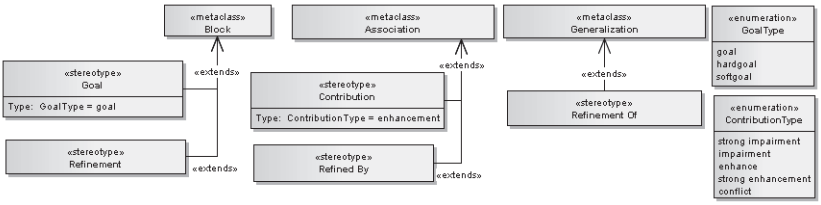


Abbildung 5: Definition von Stereotypen zur Zielmodellierung

5 Zusammenfassung

In diesem Artikel wurde ein strukturierter Ansatz zur Definition methodenspezifischer UML/SysML-Profile vorgestellt. Dazu wurde zunächst die wesentliche Literatur zur strukturierten Definition von UML/SysML-Profilen untersucht und ein auf der bestehenden Literatur basierender Ansatz vorgestellt. Die Anwendung des Ansatzes wurde am Beispiel des SPES 2020 Requirements Viewpoints gezeigt. Der Ansatz versetzt Entwickler in die Lage, methodenspezifische Profile entwickeln zu können, die hinsichtlich der in der Modellierungsmethode verwendeten Artefakte vollständig und hinsichtlich der Modellierungsmethode zu Grunde liegenden Konsistenzregeln korrekt sind. Ferner können Profile entwickelt werden, die syntaktisch zum UML-Metamodell sowie der zu unterstützenden Modellierungsmethode konsistent sind. Dadurch ermöglicht der in diesem Beitrag vorgestellte Ansatz eine grundlegende Werkzeugunterstützung für Anwender einer Modellierungsmethode, da dem entstandenen Profil ein Konzeptmodell der Modellierungsmethode zu Grunde liegt, welches werkzeugspezifisch unter Verwendung der für spezifische UML-Werkzeuge gängigen Verfahren implementiert werden kann. Somit leistet der in diesem Artikel beschriebene Ansatz einen wesentlichen Beitrag für die Industrieakzeptanz und industrielle Anwendbarkeit von Modellierungsmethoden, da neu entwickelte Modellierungsmethoden mit den in industriellen Werkzeugketten vorhandenen UML/SysML-Werkzeugen verwendet werden können, was bedeutet, dass keine neuen Werkzeuge, zusammen mit einer neuen Modellierungsmethode, eingeführt werden müssen.

Literaturverzeichnis

[AP08] Alanen, M.; Porres, I.: A Metamodeling Language Supporting Subset and Union Properties. Softw Syst Model 7(1), 2008, S. 103-124.

[BB12] Beetz, K.; Böhm, W.: Challenges in Engineering of Software-Intensive Embedded Systems. In (Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology, Springer, Heidelberg, 2012.

[Br12] Broy, M.; Damm, W.; Henkler, S.; Pohl, K.; Vogelsang, A.; Weyer, T.: Introduction to the SPES Modeling Framework. In (Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology, Springer, Heidelberg, 2012.

[Da06] Davis, I.; Green, P.; Rosemann, M; Idulska, M.; Gallo, S.: How do practitioners use conceptual modeling in practice? Data & Knowledge Engineering 58, 2006, S. 358-380.

[Da12] Daun, M.; Tenbergen, B.; Weyer, T.: Requirements Viewpoint. In (Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology, Springer, Heidelberg, 2012.

[FP10] Fowler, M.; Parsons, R.: Domain Specific Languages. Addison-Wesley Longman, Amsterdam, 2010.

[FV04] Fuentes-Fernándes, L.; Vallecillo-Moreno, A.: An Introduction to UML Profiles. Upgrade 5(2), 2004, S. 6-13.

[GLT03] Graaf, B.; Lormans, M.; Toetenel, H.: Embedded Software Engineering: The State of the Practice. IEEE Softw 20(6), 2003, S. 61-69.

[KT08] Kelly, St.; Tolvanen, J.-P.: Domain-specific Modeling - Enabling Full Code Generation. John Wiley & Sohns, New Jersey, 2008.

[La08] Lagarde, F.; Espinoza, H.; Terrier, F.; André, Ch.; Gérard, S.: Leveraging Patterns on Domain Models to Improve UML Profile Definition. In (Luiz, J.; Inverardi, P. Hrsg.): Proc. Fundamental Approaches to Software Engineering. Springer, Heidelberg, 2008.

[La09] van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley & Sohns, New Jersey, 2009.

[LPR93] Lubars, M.; Potts, C.; Richter, C.: A review of the state of the practice in requirements modeling. In: Proc. IEEE Int. Symp. Requirements Engineering, 1993.

[MOF06] Object Management Group: Meta Object Facility Version 2.0, OMG Document Number formal/2006-01-01, 2006.

[Se07] Selic, B.: A Systematic Approach to Domain-Specific Language Design Using UML. In: Proc. 10th IEEE Int. Symp. Object and Component-Oriented Real-Time Distributed Computing, 2007.

[STP12] Sikora, E.; Tenbergen, B.; Pohl, K.: Industry needs and research directions in requirements engineering for embedded systems. In: Requirements Engineering 17(1), 2012, S. 57-78.

[WS07] Weisemöller, I.; Schürr, A.: A Comparison of Standard Compliant Ways to Define Domain Specific Languages. In: Proc. Models in Software Engineering, Springer, 2007.

A Design Space Exploration Framework for Model-Based Software-intensive Embedded System Development

Matthias Büker, Stefan Henkler, Stefanie Schlegel, Eike Thaden

bueker@offis.de, henkler@offis.de, schlegel@offis.de, thaden@offis.de

Abstract: We propose an abstract framework for Design Space Exploration (DSE) in the context of model-based embedded system development. The goal is to enable the integration of a set of concrete DSE methods addressing different system characteristics and design goals while still having a common understanding of design artifacts.

1 Introduction

For the model-based design of complex embedded systems a structured and well-defined engineering process is essential to guarantee high quality products that fulfill all requirements of the stakeholders. In the project *Software Plattform Embedded Systems* (SPES) 2020[1] the *SPES Matrix* was proposed as part of the SPES Modeling Framework, which enables to handle the complexity of such systems by introducing different viewpoints and abstraction layers to model the system under development [Po12, Chap. 3]. An important step to obtain a system implementation is the *Design Space Exploration* (DSE), where design decisions are taken based on goals and requirements defined in previous phases.

We propose an abstract DSE process framework that is embedded into the SPES Matrix and aims at allocating a functional model to a distributed hardware architecture. The set of valid solutions may be restricted by constraints, which could be derived from previously defined requirements considering e.g. aspects like safety and real-time. Valid solutions are rated with respect to defined goals to facilitate a decision which candidate should be chosen as desired system implementation. Based on this abstract framework, concrete DSE methods can be implemented addressing different kinds of DSE problems that are relevant in industrial practice. This allows the user to choose from a set of methods to tackle a DSE problem where all methods are compatible to each other because they have a common understanding of system artifacts and their relations. This enables, in contrast to existing DSE approaches, to define a seamless Design Space Exploration methodology.

Some examples for concrete methods we plan to integrate are the following: In [SHL10] a DSE process is proposed based on formalized model transformation rules. These rules are derived from design constraints limiting the possible solution space and allow a mechanized exploration of possible design alternatives. Another work [KH08] is based on a synchronous language named *COLA* and presents a way to deploy clusters (tasks) on

[1]Funded by the Ministry of Education and Research (BMBF), http://spes2020.informatik.tu-muenchen.de/

a multi-processor platform. In [Bü11a; Bü11b], we proposed a concrete DSE method addressing the allocation of automotive software-applications on hierarchical distributed hardware architecture.

In the next section, we give an overview of the relevant concepts of the SPES Matrix. In Section 3, we introduce the proposed abstract Design Space Exploration Framework. Finally, a short summary of the paper is given in Section 4.

2 The SPES Matrix

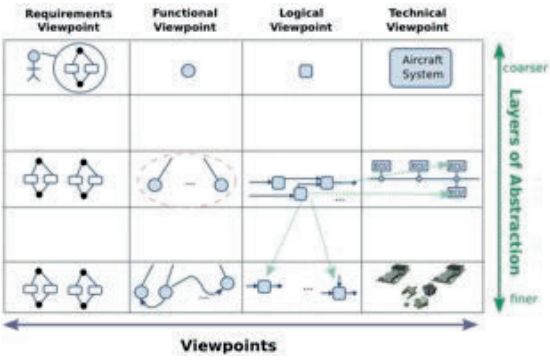


Figure 1: Matrix of Abstraction Layers and Viewpoints

To handle the complexity of embedded systems development it is useful to focus on certain aspects of the system in different development phases. In the project SPES2020, two concepts to reach this goal were introduced namely *abstraction layers* and *viewpoints* as depicted in Figure 1. Abstraction layers allow to model system artifacts on different granularity and level of detail. Viewpoints regard the same system from different perspectives while focusing on certain aspects as functional or technical concerns. In SPES2020 the following four different viewpoints were defined.

Requirements Viewpoint: To support the requirements engineering process the *Requirements Viewpoint* is introduced. It is intended to cover the relations of the system under development and its context such as users, stakeholders and external systems. To achieve this, goals and requirements are derived that the system under development should satisfy.

Functional Viewpoint: In the *Functional Viewpoint* the functions, the system under development should offer, are modeled and its syntactical interface is specified. Typically, functions arise from the requirements and goals defined on the Requirements Viewpoint. These functions may be decomposed into sub-functions and their behavior may be specified as well as their interaction with other functions or sub-functions.

Logical Viewpoint: The *Logical Viewpoint* is intended to specify a structural decomposition of the system under development into logical components independently from any

technological aspects. For this purpose, the functions identified on the Functional Viewpoint are mapped to logical components by means of trace links. The interaction of logical components is modeled by logical channels and an interface is defined, which the system offers to communicate with its context. Nevertheless, the Logical Viewpoint may still model the functionality of the system but now realized in a logical architecture.

Technical Viewpoint: The *Technical Viewpoint* describes the physical architecture of the system in terms of hardware resources in combination with its software parts in terms of tasks. By means of trace links a deployment mapping is specified defining on which hardware resources tasks of the Logical Viewpoint are executed and how logical subsystems are realized. Important aspects that are treated on this viewpoint are the specification and verification of timing and safety properties such as resource consumption and redundancy.

3 Design Space Exploration Framework

In the context of this work, *Design Space* refers to all configurations of a given embedded hardware/software system satisfying a set of requirements and constraints. *Design Space Exploration* refers to the process of systematically searching a Design Space for (near-) optimal solutions with respect to one or more given optimization objectives. In Figure 2 the proposed framework for DSE is shown and described in the following.

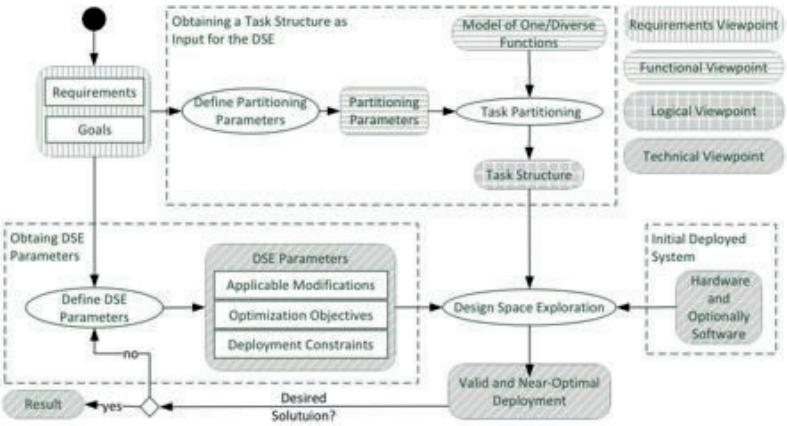


Figure 2: Overview of Design Space Exploration Framework

3.1 Defining the Design Space

Initial Deployed System: Developing embedded systems is typically done incrementally. This means that systems are built on top of an existing predecessor model, like for exam-

ple a model line of a car. This is what we refer to as *Initial Deployed System*. Such a system is built of Electronic Control Units (ECUs) with some already allocated tasks. The overall capacity of all ECUs, less the needed capacity for the already allocated tasks, is the available resource for task allocation. At this point, we explicitly differ between software and hardware parts of the system, which is a characteristic of the Technical Viewpoint.

Task Structure: New functions usually are developed with the help of some design tool, like for example Matlab Simulink. In such tools the system is modeled primary with respect to functional aspects. That is why such a model is related to the Functional Viewpoint. To be able to allocate functions to ECUs we need to define atomic processes (tasks) that are to be executed on hardware resources. Thus, the functional model is partitioned into tasks under logical aspects, which are typically different from the functional decomposition. Such logical aspects might be, for example, communication dependencies between functions or safety constraints. The result of this *Task Partitioning* process is a logical task structure, which serves as input for the succeeding DSE step.

The Influence of the Requirement Viewpoint: In the Requirements Viewpoint goals and requirements are specified, which are refined to constraints and objectives relevant for DSE. On the one hand, these might be constraints that influence the Task Partitioning process in terms of *Partitioning Parameters*. Such a constraint might, for example, forbid that two functions are mapped to the same task. An example for an objective is that communication between tasks should be minimized.

On the other hand, the Requirements Viewpoint also influences the DSE itself via *DSE Parameters* in terms of constraints, objectives, and applicable modifications. An example for a relevant goal might be that the system costs should be low leading to the objective to minimize costs. A safety requirement, for example, might lead to a constraint stating that some tasks are not allowed to get allocated to the same ECU. Modifications allow modifying the hardware architecture by adding new ECUs or exchanging existing ones.

3.2 Design Space Exploration

In the DSE process the given task structure should be allocated to the Initial System while considering all applicable modifications, satisfying all constraints and optimizing the given objectives. Because of the high complexity of the optimization problem often heuristic methods are used to search the Design Space leading to near-optimal solutions.

The described abstract DSE framework may be realized by different concrete DSE methods that are refinements of the abstract process. Refinement means here, for example, that the DSE step may be detailed in terms of different phases describing a concrete DSE method. Furthermore, the DSE Parameters need to be concretized by describing what kind of DSE Parameters are supported by a certain method and how they are extracted from the Requirement Viewpoint. To be able to rate solutions with respect to an optimization goal, this goal needs to be formalized to an optimization function.

3.3 DSE Results and Iterations

For a solution to be valid all tasks have to be allocated and yield in a feasible system whereas all given constraints are satisfied. In this case complete solutions exist, and the DSE returns the best solutions for the optimal deployment problem with respect to the optimizing goal. Depending on the optimization method such solutions might be optimal or near optimal. If there are several (near-) optimal solutions the user can decide whether one of them fits his needs. If this is the case, the process is finished. Otherwise, the DSE Parameters may be adjusted to guide the process into the desired direction. If no valid solution exists, the DSE returns an incomplete result whereas some tasks got allocated. In this case it is possible to modify the DSE Parameters and run the DSE step again to allocate the remaining tasks and to obtain a valid solution. Changes of the DSE Parameters may be valid refinements of the linked requirements but they might also be changes that have influences to the Requirement Viewpoint. Thus, it might be necessary to adapt requirements such that the refinement relation is restored. Based on these changes the DSE may be executed again in another iteration to obtain further variants of the deployed system.

4 Conclusion

We presented an abstract DSE framework embedded into the concepts of viewpoints and abstraction layers of the SPES Matrix. This DSE framework allows the integration of a set of concrete DSE methods with well-defined interfaces to address different DSE problem characteristics.

References

[Bü11a] Büker, M. et al.: An Automated Semantic-Based Approach for Creating Tasks from Matlab Simulink Models. In: Proc. of 16th International Workshop on Formal Methods for Industrial Critical Systems (FMICS). 2011.

[Bü11b] Büker, M. et al.: Automating the Design Flow for Distributed Embedded Automotive Applications: Keeping Your Time Promises, and Optimizing Costs, too. In: Proc. International Symposium on Industrial Embedded Systems (SIES). 2011.

[KH08] Kugele, S.; Haberl, W.: Mapping Data-Flow Dependencies onto Distributed Embedded Systems. In: Proc. of SERP 2008. 2008.

[Po12] Pohl, K. et al.: Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology. In: Berlin Heidelberg: Springer, 2012. ISBN: 978-3-642-34613-2.

[SHL10] Schätz, B.; Hölzl, F.; Lundkvist, T.: Design-Space Exploration through Constraint-Based Model-Transformation. In: Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems. ECBS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 173–182. ISBN: 978-0-7695-4005-4.

Anforderungen von Leitbranchen der deutschen Industrie an Variantenmanagement und Wiederverwendung und daraus resultierende Forschungsfragestellungen*

Martin Große-Rhode[1], Peter Manhart[2], Ralf Mauersberger[3], Sebastian Schröck[4], Michael Schulze[5], Thorsten Weyer[6]

[1]Fraunhofer FOKUS
Steinplatz 2
10623 Berlin
martin.grosse-rhode@
fokus.fraunhofer.de

[2]Daimler AG
Wilhelm-Runge-Straße 11
89081 Ulm
peter.manhart@
daimler.com

[3]EADS Innovation Works
81663 München
ralf.mauersberger@
eads.net

[4]Helmut-Schmidt-Universität
Holstenhofweg 85
22043 Hamburg
sebastian.schroeck@
hsu.hh.de

[5]pure-systems GmbH
Agnetenstr. 14
39106 Magdeburg
michael.schulze@
pure-systems.com

[6]Universität Duisburg-Essen
Gerlingstraße 16
45127 Essen
thorsten.weyer@
paluno.uni-due.de

Abstract: Im Rahmen des bereits abgeschlossenen BMBF-Projektes SPES 2020 wurden mit dem SPES 2020 Modeling Framework verschiedene integrierte Konzepte, Techniken und Methoden zum durchgängigen Engineering von softwareintensiven eingebetteten Systemen entwickelt. Die aus Sicht der Industrie äußerst relevante Fragestellung nach einer möglichst bruchfreien Unterstützung des durchgängigen Variantenmanagements und der Wiederverwendung wurde dabei bewusst nicht betrachtet. Im Rahmen des Nachfolgeprojektes SPES_XT soll nun der SPES 2020 Modeling Framework um die Unterstützung für ein durchgängiges Variantenmanagement und der Wiederverwendung erweitert werden. Im vorliegenden Beitrag werden hierzu die systematisch erarbeiteten Anforderungen der Branchen Automatisierungstechnik, Automotive und Avionik vorgestellt. Auf Grundlage der Anforderungen werden dann zentrale Forschungsfragestellungen skizziert.

1. Motivation

Produkte aus den Branchen Avionik, Automatisierung und Automotive sind gekennzeichnet durch eine enorme Anzahl verschiedenster Bestandteile wie mechanische Komponenten, Aktuatorik, Sensorik, Kabel, Leitungen, etc. und sogenannten Embedded Systems, bestehend aus Hardware und Software. Die Entwicklung solcher komplexen Gesamtsysteme ist eine höchst anspruchsvolle Aufgabe im Engineering, zumal der Umfang der Systeme stetig zunimmt und dieser Trend sich eher verstärkt denn abschwächt. Ferner erbringen oft nicht einzelne Systeme sondern Systemverbünde eine Aufgabe in Kooperation und deren korrektes Zusammenwirken stellt ebenfalls eine vielschichtige und komplizierte Herausforderung dar. Der Ursprung neuer innovativer, für den Kunden

* Dieser Beitrag wurde durch das BMBF im Projekt SPES 2020_XT (Förderkennz: 01IS12005) gefördert.

erlebbarer, Funktionen liegt oft im Bereich der Embedded Systems, die somit maßgebend für den Fortschritt und zukünftige Marktchancen verantwortlich sind.

Neben der Komplexität aufgrund der herausfordernden Aufgabenstellung wird diese weiterhin durch regulatorische Vorschriften unterschiedlicher anvisierter Märkte als auch durch kundenspezifische Wünsche erhöht. Jene Anforderungen bedeuten Variabilität sowohl in der Hardware als auch in der Software. Als immanente Eigenschaft der Embedded Systems erschwert diese Variabilität als zusätzlicher Freiheitsgrad also die Entwicklung. Die Beherrschung der gesamten Entwicklung und sämtlicher Freiheitsgrade ist daher entscheidend im Wettbewerb mit Blick auf die Time-to-Market, die Qualität, die Kostenstrukturen und damit den Erfolg des Unternehmens am Markt. Unzulänglichkeiten während der Entwicklung führen teils zu finanziellen Schäden, Imageverlusten oder gar zu Gefahren für die körperliche Unversehrtheit von Menschen.

Die Komplexität der Embedded Systems, hervorgerufen durch die eigentlich zu lösende Aufgabenstellung, lässt sich prinzipienbedingt nicht minimieren sondern nur mit einem „teile und herrsche“ Ansatz begegnen, der die Gesamtaufgabe in kleinere handhabbarere Teilaufgaben zerlegt. Den Herausforderungen der Variabilität und den wettbewerbsrelevanten Faktoren marktgerechte Kosten, hohe Produktqualität und kurze Time to Market kann dagegen mit geeigneten Mitteln wie dem Variantenmanagement auf der einen Seite und der methodischen und systematischen Wiederverwendung auf der anderen begegnet werden. In Kombination beider Mittel kann deren Potential vollumfänglich genutzt und Mitnahmeeffekte, wie Aufwands- und Kostenreduktion sowie Qualitätssteigerungen über unterschiedliche Varianten hinweg, erzielt werden. Solche Synergieeffekte werden in der heutigen Praxis jedoch bei weitem noch nicht ausgeschöpft.

In Abschnitt 2 werden die wesentlichen branchenspezifischen Anforderungen vorgestellt. Anschließend werden in Abschnitt 3 diese Anforderungen analysiert. In Abschnitt 4 werden dann die zugehörigen Forschungsfragestellungen des durchgängigen Variantenmanagements und der systematischen Wiederverwendung erläutert.

2. Anforderungen aus den Leitbranchen

Im Rahmen der Anforderungserhebung wurden zunächst 60 Use Cases dokumentiert. Daraus wurden 137 Anforderungen abgeleitet und klassifiziert. In den folgenden drei Abschnitten werden wesentliche Anforderungen der Branche zusammengefasst. Dem folgt jeweils die Darstellung eines zentralen Use Case.

2.1 Anforderungen aus Branche „Automatisierungstechnik“

In der Domäne Automatisierungstechnik wird in SPES_XT das Engineering automatisierter Anlagen betrachtet. Folglich unterscheidet sich diese Domäne von den anderen betrachteten Branchen vor allem im Hinblick auf das *System-under-Development*, das Geschäftsmodell sowie die zu spezifischen Randbedingungen. Eine automatisierte Anlage wird i.d.R. nur einmal kundenspezifisch projektiert und errichtet, sodass die Kosten des Engineerings nicht auf große Stückzahlen umgelegt werden können. Des Weiteren sind am Engineering eine Vielzahl von Gewerken beteiligt, weshalb hier im Besonderen die Wirkbeziehungen zwischen den interdisziplinären Artefakten zu berücksichtigen sind, um eine systematische Wiederverwendung zu etablieren [AAF03].

Die Use-Cases und die daraus abgeleiteten Anforderungen decken das gesamte Engineering ab und lassen sich in phasenübergreifende und phasenspezifische Use-Cases unterteilen. Als zentraler Use-Case ist hier die *Wiederverwendung gewerkeübergreifender Artefakte unter Berücksichtigung aller Wirkbeziehungen und Restriktionen* herauszustellen. Hierbei soll nicht nur die Software wiederverwendet werden, sondern auch Artefakte vorgelagerter Gewerke. Tabelle 1 zeigt den Ablauf dieses Soll-Use-Cases.

Schritt	Akteur	Ablauf
1	Verfahrenstechnik / Prozessleittechnik	*Auffinden des Artefaktes*: Ein für die Wiederverwendung vorgesehenes gewerkeübergreifendes Artefakt wird anhand der Anforderungen aufgefunden und für die Wiederverwendung herangezogen. Beispielhaft sei hier ein Anlagenteil. Die Auswahl geschieht in einer frühen Phase, in der das Verfahren und nicht etwaige Software adressiert wird.
2	Verfahrenstechnik / Prozessleittechnik	*Identifikation verbundener Artefakte anderer Gewerke:* Das exemplarische Modell einer Teilanlage ist evtl. untergliedert in weitere Artefakte, die z.B. Mechanik, Steuerungshardware und Leistungselektronik repräsentieren. Hier müssen alle Wirkbeziehungen zwischen Gewerken und entsprechenden Softwarefragmenten berücksichtigt werden.
3	Automatisierungstechnik	*Wiederverwendung gewerkeübergreifender Artefakte*: Gemäß den Anforderungen im Projekt, werden Artefakte wiederverwendet oder individuell ergänzt. Dies gilt gerade auch für die Automatisierungssoftware, welche nun in ein vorgegebenes Umfeld aus Mechanik und Elektronik eingepasst werden muss. Eine geeignete und durchgängige Softwareunterstützung ist hierfür unerlässlich.

Tabelle 1: Soll-Szenario des UC „Wiederverwendung eines gewerkeübergreifenden Artefakts“

Die Herausforderung, die der angeführte Use-Case in allgemeiner Form beschreibt, ist, dass die Entwicklung der Automatisierungssoftware für eine nahezu vollständig ausgeplante Anlage zu geschehen hat. Diese Aufgabe findet folglich in einem Spannungsfeld aus Zeit- und Kostendruck, erheblichen Anforderungen bezüglich der Effizienz und Qualität, aber vor allem auch unter der Einschränkung der bereits definierten Anlage statt. Hier ist es unerlässlich geeignete Konzepte zu schaffen und die Softwaretools dahingehend weiterzuentwickeln, dass diese wiederverwendungsorientiertes Engineering unterstützen. Die Herausforderung hierbei ist es die inhaltliche Durchgängigkeit über die verschiedenen Gewerke und Phasen zu gewährleisten. Um dieser Heterogenität zu begegnen sind optimierte und angepasste Methoden für das Engineering nötig.

2.2 Anforderungen aus der Branche „Automotive“

Die Anforderungen der Branche „Automotive“ konzentrieren sich auf die Herausforderungen des durchgängigen Variantenmanagements. Wir verstehen unter durchgängigem Variantenmanagement die konsistente und vollständige Erfassung von Variationspunkten in allen Entwicklungsartefakten (Anforderungsdokumente, Funktionsmodelle, Sicherheitsanalysen, Testbeschreibungen, Parameterkonfigurationen, usw.) und deren Abbildung in ein integrierendes Varianten- und Konfigurationsmodell. Die entsprechenden Anforderungen lassen sich in die Kategorien (1) Reengineering bestehender Varianten, (2) Produktlinienentwicklung und (3) Sonstige Anforderungen einteilen. Im Rahmen des Reengineering wird eine Menge von Realisierungen zu einer Produktlinie zusammengeführt. Diese Struktur wird dann im Rahmen der Produktlinienentwicklung kontinuierlich weiterentwickelt. Sonstige Anforderungen, wie z.B. Konformität mit Standards wie ISO 26262 oder AUTOSAR, wurden in Kategorie (3) eingeordnet.

Der aus Sicht der Domäne Automotive zentrale Use-Case bei durchgängigem Variantenmanagement ist *die konsistente Umsetzung eines Änderungsantrages im Laufe der*

Produktevolution. Im Verlauf der Bearbeitung wird für mehrere Artefakte zunächst das eigentliche Artefakt und dann dessen Konfigurationsmodell aktualisiert. Diese Kette wird in dem folgenden Soll-Use-Case abgekürzt dargestellt (vgl. Tabelle 2, Tabelle 3):

Schritt	Akteur	Ablauf:
1	*Antragsteller*	*Beschreibt* einen neuen Change-Request inklusive der Systemausprägungen (Varianten aus Systemsicht/Kunde: Ausstattungslinie, Motor, Baureihe)
2	*Projektverantwortlicher*	*Analysiert* den neuen CR und ermittelt die Merkmale, welche die Systemausprägungen charakterisieren, Ermittelt die Releases in denen der CR zu tragen kommt
3	*Requirements Engineer*	Passt das *Lastenheft an* indem er eine neue Ausprägung hinzufügt oder einen neuen Variationspunkt erstellt.
4	*Variantenmodellierer*	Aktualisiert das *Konfigurationsmodell* des Lastenhefts
5+6	Testentwickle + Variantenmodellierer	Aktualisiert die *Testspezifikation* und deren *Konfigurationsmodell.*
7+8	*Funktionsmodellierer +Variantenmodellierer*	Aktualisiert die *Implementierung* und dessen *Konfigurationsmodell*
9+10	*Testteam + Variantenmodellierer*	Aktualisiert die *Modultests* und deren *Konfigurationsmodell*
11+12	*Softwareintegrator*	*Passt Integrationsregeln* an und ggf. den *Build-Prozess an*
13+14	*Testteam + Variantenmodellierer*	Aktualisiert die *HIL- und Systemtests* und deren *Konfigurationsmodell*
15+16	*Applikateur + Variantenmodellierer*	*Kalibrierung* der geänderten Funktion und erstellt ggf. neue Datensätze und deren *Konf.Mod.*
17	*Antragsteller*	*Abnahme* der Änderung für alle Varianten, in denen der CR umgesetzt wurde

Tabelle 2: Soll-Szenario des UC „Umsetzung eines Änderungsantrages in der Produktevolution“

Ausnahmen, Erweiterungen		
2.a	*Projektver-antwortlicher*	Falls zusätzliche Merkmale nötig sind, wird der CR um entsprechende Hinweise ergänzt
2.b	*Varianten-modellierer*	*Aktualisiert* das *Merkmalmodell* inklusive der Beziehungen entsprechend des CR
3.a	*Safety Engineer*	Erstellt eine Sicherheitsanalyse (z.B. FTA oder FMEA) falls das hinzugefügte Merkmal sicherheitsrelevant ist bzw. wiederholt jene Analyse.
4.a/6.a/8.a/10.a /12.a/14.a/ 16.a	*Varianten-modellierer*	Unter Umständen muss das *Merkmalsmodell* ebenfalls geändert werden

Tabelle 3: Ausnahmen und Erweiterungen des Soll-Szenarios im UC aus Tabelle 2

Dieser Umsetzung dieses Use-Case ist zweifelsohne anspruchsvoll, eine adäquate Umsetzung verspricht allerdings auch ein hohes Maß an Konsistenz von Variantenmodell und Entwicklungsartefakten in der evolutionären Entwicklung und eine wertvolle Grundlage für Traceability auf Basis gemeinsamer Merkmale von Artefakten. Darum wollen wird dieser Use-Case im Projekt SPES_XT mit hoher Priorität bearbeiten.

2.3 Anforderungen aus Branche „Avionik“

Konventionelle Avioniksysteme in Luftfahrzeugen sind gekennzeichnet durch eine Vielzahl unabhängiger Teilsysteme, die ganz bestimmte Funktionalitäten wahrnehmen und

auch nur für diese konzipiert und zugelassen sind. Die Software solcher proprietären *Federated Systems* ist hierbei sehr stark abhängig von der zugrundeliegenden Hardware, sodass bereits geringfügige funktionale Änderungen oder Erweiterungen des Systems zu erheblichen Anpassungs- und damit Re-Zertifizierungsaufwendungen führen. Daher werden seit einigen Jahren Anstrengungen unternommen, diese Eigenschaften im Rahmen einer *Integrierten Modularen Avionik (IMA)* zu verbessern. Die Flexibilität verteilter Systemarchitekturen soll auch im Bereich der Avionik zum Tragen kommen. Ähnlich einem Rechnernetzwerk werden daher Rechnermodule unter Verwendung einer standardisierten Software-Architektur und eines Echtzeit-Betriebssystems miteinander vernetzt, sodass Funktionen modulübergreifend gestartet und betrieben werden können.

Um systematische Wiederverwendung bei der Entwicklung von IMA-Systemen zu berücksichtigen, ist die Granularität der Wiederverwendung entscheidend. Eine Granularität unterhalb der Echtzeit-Prozess-/Thread-Ebene gilt als kompliziert, sodass kaum Vorteile zu erwarten sind. Die Wiederverwendung zielt daher auf diese Ebene, wobei eine Funktionalität mehrere Threads (sog. Building Blocks) umfassen kann.

Der aus Sicht der Domäne Avionik zentrale Use-Case für durchgängiges Variantenmanagement und systematische Wiederverwendung ist die *konsistente Assemblierung eines IMA-Systems aus wiederverwendbaren Building Blocks* nach den oben dargestellten Kriterien. Tabelle 4 zeigt den Ablauf des Soll-Use-Cases.

Schritt	Akteur	Ablauf:
1	*Requirements Engineer*	Spezifiziert funktionale IMA Requirements
2	*Requirements Engineer*	Spezifiziert nicht-funktionale IMA Requirements, wie z.B. Ausfallwahrscheinlichkeiten, Performanz
3	*Systems Engineer*	Definiert Architektur des IMA Systems. Trennung von Hardware und Software. Basierend auf initialer Architektur werden Ausfallwahrscheinlichkeiten abgeschätzt
4	*Software Engineer*	Simuliert Modell des IMA Systems auf logischer Ebene
5	*Software Engineer*	Simuliert Modell des IMA Systems auf Kommunikations-Ebene
6	*Software Engineer*	Definiert IMA System. Definiert wiederverwendbare Building Blocks. Legt alle Parameter des Schedulers fest
7	*Software Engineer*	Implementierung und Assemblierung des IMA Systems
8	*Test Engineer*	Testen der Pfadabdeckung, Testen der Requirements, Testen der nicht-funktionalen Requirements
9	*Safety Engineer*	Nachweis der Req.-Traceability in beide Richtungen. Nachweis der Abhängigkeiten. Anfertigen Fehlerbäume, FMEA Analyse
10	*Airworthiness Autorities*	Zertifizierung des IMA Systems

Tabelle 4: Soll-Szenario des UC „konsistente Assemblierung eines IMA-Systems“

Wiederverwendbare Entwicklungsartefakte für IMA müssen die in obigem Use Case beschriebenen Eigenschaften erfüllen. Natürlich sollte dabei eine Assemblierung der Building Blocks die Architektur des Systems nicht grundlegend verschlechtern. Der Safety-Aspekt sollte durch systematische Wiederverwendung gestärkt werden. Keinesfalls darf ein System dadurch unsicherer werden. Die entsprechenden Nachweisverfahren sind weiterhin zu erbringen. Ferner sollte Wiederverwendung im Kontext von IMA

Kosten und Zeit sparen. Hierzu sind die benötigten Techniken und Werkzeuge zu realisieren und zu definieren, wie jene im Engineering zu platzieren sind.

3. Gemeinsamkeiten und Unterschiede der Anforderungen

Vergleicht man die Anforderungen aus den drei Branchen, so fallen zunächst die Unterschiede auf. Automatisierungstechnik und Avionik stellen Anforderungen in Bezug auf die Wiederverwendung, aus der Automobilbranche kommen Anforderungen an das Variantenmanagement. Der jeweils andere Aufgabenbereich wird durch die hier getroffene Auswahl der wichtigsten und dringlichsten Anforderungen ausgeschlossen.

3.1 Unterschiede

Im Hinblick auf das Produktangebot ist die explizite Unterscheidung von Variantenmanagement und Wiederverwendung evident. In der Automobilindustrie müssen sehr viele, einander sehr ähnliche Produkte quasi gleichzeitig angeboten werden. Das gilt nicht nur für die Fahrzeughersteller, die – u.a. – Kundenwünsche nach individueller Konfiguration umgehend bedienen können müssen, sondern auch für Zulieferer, die mit vielen Herstellern zusammen arbeiten. In den Branchen Automatisierung und Avionik hingegen werden Entwicklungsprojekte beinahe produktspezifisch aufgesetzt, da die Systeme größer, die Stückzahlen geringer und die Entwicklungszyklen länger sind. Betrachtet man nicht nur die Avionik sondern die Flugzeugentwicklung, so findet man kundenspezifische Anforderungen, die ein gewisses Maß an Variantenmanagement nahelegen.

Mit der Unterscheidung von Volumenproduktion und Variantenmanagement auf der einen und Individualproduktion und Wiederverwendung auf der anderen Seite gehen offensichtlich – zumindest in den hier erfassten Anforderungen – auch unterschiedliche technische Entwicklungsstände bzw. Prozessreifegrade in Bezug auf die Themen Wiederverwendung und Variantenmanagement einher. Der Automotive-Anwendungsfall nimmt nicht nur sehr konkret Bezug auf differenzierte Prozessschritte und Rollen im Variantenmanagement. Die Verwendung von Begriffen wie Merkmals- und Konfigurationsmodell zeigt, dass hier bereits mit Werkzeugen gearbeitet wird, die auf eine modellbasierte Softwareentwicklung aufsetzen und diese um die Modellierung und Verwendung von Merkmalen erweitern. Aus dem Use Case und dessen Voraussetzungen können daher unmittelbar Anforderungen und Forschungsfragen abgeleitet werden.

Für die Anwendungsfälle aus den anderen beiden Branchen sind die Prozessschritte und Werkzeuge nicht ausdifferenziert. Neben der unterschiedlichen Struktur des Produktportfolios spielen technische Gegebenheiten, wie die verwendeten Programmiersprachen und das Zusammenspiel verschiedener Gewerke, rechtliche Randbedingungen und nicht zuletzt wirtschaftliche bzw. kulturelle Gegebenheiten der Branchen, eine Rolle. Vor dem Hintergrund der Erkenntnisse aus der Softwaretechnik zur Wiederverwendung ist hier also nach den Voraussetzungen zu fragen, die erfüllt sein müssen, um die in der Literatur beschriebenen Ansätze zur Wiederverwendung anwenden zu können.

3.2 Voraussetzungen für die Wiederverwendung

Die zentrale Frage des Anwendungsfalls aus der Automatisierungstechnik ist, wie bestehende Software effizient und sicher an geänderte Voraussetzungen – aus den anderen

Gewerken – angepasst werden kann. Das zielt zunächst auf die Dokumentation der Abhängigkeiten ab, die zwischen der bestehenden Software-Version und ihrem Umfeld (Hardware etc.) bestehen. Je besser diese bekannt und analysierbar sind, desto besser und effizienter kann der Änderungsbedarf in der Software benannt werden. Hierbei wird Bezug genommen auf die Automatisierungs-Software, die das Anlagenverhalten steuert.

Implizit wird durch den Anwendungsfall aber auch die Besonderheit der Software in der Automatisierungstechnik angesprochen: Da die Softwareentwicklung eine der letzten Tätigkeiten des Engineerings automatisierter Anlagen ist, sind die Randbedingungen der Software wenig flexibel. Unabhängigkeit von Entwicklungsprozess, Hardware, und Kommunikationsmitteln wird bisher nicht durch spezielle Techniken der Wiederverwendung erreicht. Der wesentliche Punkt hierbei ist zunächst die Modularisierung der Software. In der Automatisierungstechnik ist die Entwicklung der Software von Programmiersprachen wie denen der IEC 61131-3 geprägt, die keine Modularisierungskonzepte für eine architekturzentrierte Entwicklung anbieten.

Im Anwendungsfall der Avionik ist eine Architektur, die Plattform und Anwendung trennt, vorausgesetzt. Sie ermöglicht Wiederverwendung, stellt aber auch strenge Bedingungen an die Anwendungssoftware, die für diese Architektur entwickelt werden kann. Als Einheit der Wiederverwendung werden Prozesse oder Threads benannt, ohne auf einen inhaltlichen Kontext Bezug zu nehmen. Weitergehende Fragen zur systematischen Wiederverwendung müssen genau diesen Kontext adressieren.

Die bisherige Analyse der Anwendungsfälle und Anforderungen scheint darauf hinaus zu laufen, dass die Domänen einzeln behandelt werden müssen. Die Unterschiede sind groß, Gemeinsamkeiten waren kaum auszumachen. Auch eine Betrachtung aller Anwendungsfälle und Anforderungen, die im Projekt SPES-XT erhoben wurden, schwächt diesen Eindruck nicht wesentlich ab. Gemeinsamkeiten finden sich am ehesten bei Anforderungen, die gezielt abstrakt formuliert wurden, um ggf. übertragbar zu sein. Allerdings stellt sich bei diesen Anforderungen die Frage, inwiefern das Abstraktionsniveau noch hilfreich für die beabsichtigten Anwendungen bei den Industriepartnern ist. Die Identifikation von Gemeinsamkeiten liegt eher im Interesse der Forschungspartner, sowohl in der Grundlagen- als auch in der Anwendungsforschung. Selbst wenn ein unmittelbares Übertragen von Lösungen angesichts der Heterogenität der Branchen nicht sehr realistisch erscheint, wäre doch das Herausarbeiten von gemeinsamen Prinzipien langfristig hilfreich; für die Grundlagenforschung im Hinblick auf die Formulierung von Theorien und Modellen und für die Anwendungsforschung im Hinblick auf ein Beratungsangebot, das auf einer profunden und umfassenden Analyse beruht.

Implizite Gemeinsamkeiten wurden oben bereits genannt, in Form der Voraussetzungen, die für Variantenmanagement und / oder Wiederverwendung erfüllt sein müssen: modellbasierte architekturzentrierte Entwicklung, Modularität, technische Anschlussfähigkeit (d.h. geeignete Programmiertechnologien oder Generatoren). Auch eine gleichzeitige Betrachtung von Variantenmanagement und Wiederverwendung auf einer sehr abstrakten Ebene wird zumindest Ähnlichkeiten zeigen, zum Beispiel im Hinblick auf Kontext- und Architekturmodellierung, Modellverwaltung, die Integration von qualitätssichernden Maßnahmen und den Zusammenhang mit dem Life Cycle Management.

4. Forschungsfragestellungen

Wie in Abschnitt 3 gezeigt, besitzen die drei betrachteten Branchen in Bezug auf ein durchgängiges Variantenmanagement und Wiederverwendung teils sehr spezifische Anforderungen, die von Konzepten, Techniken und Methoden erfüllt werden müssen, um die einzelnen Use Cases unterstützen zu können. Wenngleich zwischen den Branchen eine teils hohe Heterogenität bzgl. der betrachteten Fragestellungen herrscht, besteht für sämtliche zu entwickelnden Ansätze die zentrale Forderung, dass diese sich bruchfrei in den bestehenden SPES 2020 Modeling Framework [Br+12] integrieren.

Sowohl für das Gebiet durchgängiges Variantenmanagement, welches in erster Linie von den Partnern innerhalb der Branche Automotive betrachtet wird, als auch für das Gebiet systematische Wiederverwendung, das im Projekt von Partnern der Branchen Automatisierungstechnik und Avionik betrachtet wird, gilt das die teilweise sehr unterschiedlichen Forschungsfragstellungen in grobe Kategorien unterteilt werden können.

4.1 Kategorie „Erweiterung der bestehenden SPES 2020 Modellierungstheorie“

In diese Kategorie fallen solche Forschungsfragestellungen, die sich darauf beziehen, wie die bestehende Modellierungstheorie des SPES 2020 Modeling Framework [Br+12] spezifisch um solche Konzepte erweitert werden kann, die den Umgang mit Varianten und wiederverwendbaren Artefakten im Entwicklungsprozess gestatten. Im Detail stellen sich u.a. die folgenden verfeinerten Forschungsfragen: (1) Welche neuen bzw. veränderten Konzepte und Traceability-Beziehungen sind notwendig um die Tätigkeiten in den einzelnen Use-Cases des durchgängigen Variantenmanagements von Seiten der Modellierung zu unterstützen? (2) Welche neuen bzw. veränderten Konzepte und Traceability-Beziehungen sind notwendig, um die Tätigkeiten in den einzelnen Use-Cases zur systematischen Wiederverwendung von Seiten der Modellierung zu unterstützen?

Im Rahmen der Ausarbeitung entsprechender Lösungen ist es notwendig die ontologische Basis eines oder mehrerer Viewpoints (vgl. [DTW12], [V+12], [EMV12], [W+12]) des SPES 2020 Modeling Frameworks konform zum IEEE Std. 1471 [IE00] bzw. dessen Nachfolger IEEE Std. 42010 [IE11] zu erweitern oder eine zusätzliche Modellierungsperspektive (z.B. Variabilität) zu definieren, die z.B. querschneidend zu den verschiedenen Viewpoints des SPES 2020 Modeling Frameworks stehen (vgl. z.B. [HK+12]) . Im letzteren Falle muss für eine bruchfreie Integration der präzise semantische Bezug der Konzepte zur jeweiligen Ontologie der SPES 2020 Viewpoints definiert werden.

4.2 Kategorie „Methodik für Variantenmanagement und Wiederverwendung“

In diese Kategorie fallen solche Forschungsfragestellungen, die auf Basis der erweiterten Modellierungstheorie des SPES 2020 Modeling Frameworks darauf abzielen, eine methodische Anleitung zu erarbeiten, die die im Engineering-Prozess beteiligten Personen bei der Durchführung von Tätigkeiten innerhalb der betrachteten Use Cases unterstützen. Im Detail stellen sich u.a. die folgenden Forschungsfragestellungen: (1) Welche methodischen Schritte sind in welchen Situationen in den einzelnen Use Cases im Variantenmanagement und Wiederverwendung notwendig, um den jeweiligen Use Case erfolgreich zu Ende führen zu können. (2) Welche Auswirkungen haben spezifische Aspekte des Prozesskontexts auf die methodischen Schritte zur Abarbeitung eines Use Cases?

Im Rahmen der Lösungsausarbeitung ist es notwendig, für sämtliche Tätigkeiten in der Methodik einen präzisen Bezug zu der erweiterten Modellierungstheorie des SPES Modeling Frameworks herzustellen. Darüber hinaus sollten für alle Tätigkeiten Vorbedingungen (d.h. der Prozesskontext) definiert sein, die erfüllt sein müssen, um Tätigkeiten erfolgreich durchzuführen. Zu jeder Anleitung zur Durchführung von Tätigkeiten werden noch Techniken und Werkzeuge angegeben, die deren Durchführung unterstützen.

4.3 Kategorie „Bewertungsverfahren für variable Entwicklungsartefakte“

In diese Kategorie fallen solche Forschungsfragen, die auf die Bewertbarkeit von Entwicklungsartefakten abzielen, die im Rahmen des durchgängigen Variantenmanagement und der Wiederverwendung der einzelnen Use Cases betrachtet werden. Im Detail stellen sich u.a. die folgenden Fragestellungen: (1) Welche Verfahren müssen entwickelt werden, um innerhalb der Use Cases des durchgängigen Variantenmanagement die Qualität ggf. unterschiedlicher Variabilitätsmodelle oder des aus der Bindung von Variabilität hervorgehenden Systemmodelles bewerten zu können? (2) Welche Verfahren müssen entwickelt werden, um innerhalb der Use Cases zur Wiederverwendung Aussagen über Eigenschaften eines komponierten Systems treffen zu können? Dies gilt z.B. für Echtzeiteigenschaften bei der Assemblierung wiederverwendbarer Komponenten.

Im Rahmen der Ausarbeitung entsprechender Lösungen müssen die Verfahren dabei einen präzisen Bezug zu den Artefakten des erweiterten SPES 2020 Modeling Frameworks haben. Die Resultate aus der Durchführung der Verfahren, wie z.B. die Ergebnisse des Echtzeitverhaltens, werden häufig einen deutlichen Bezug zu der Methodik für die Durchführung von Tätigkeiten innerhalb der Use Cases haben (vgl. [HK+12]).

4.2 Kategorie „Werkzeugtechnische Unterstützung“

In diese Kategorie fallen solche Forschungsfragen, die sich auf die werkzeugtechnische Unterstützung und Realisierung des durchgängigen Variantenmanagements und der systematischen Wiederverwendung beziehen. Drei Bereiche sind dabei von entscheidender Bedeutung, damit die anvisierten Effekte wie kurze Time-to-Market, hohe Produktqualität und marktgerechte Kosten ihr Potenzial entfalten, wobei sich im Detail folgende Fragenstellungen ergeben: 1) Lassen sich stets dieselben Variantenmodellierungskonzepte durchgängig über sämtliche Phasen der Entwicklung hinweg in den eingesetzten Werkzeugen nutzen oder besteht hier ebenfalls Variationsbedarf? 2) Welches sind die geeignetsten Darstellungsformen von Variabilität, damit jene vom Entwickler gut erkennbar und verständlich ist und wie wird er bei der Modellierung und seiner jeweiligen Rolle am besten werkzeugseitig unterstützt? 3) Wie kann Durchgängigkeit in Bezug auf automatisierte Prozesse aussehen, wo z.B. ein Variantenmanagement Teil eines automatischen möglicherweise zyklischen Prozesses ist und selbst Einfluss auf diesen ausüben kann? 4) Welche Verfahren bzgl. Analyse und Überführung von „Alt“-Software in Richtung Wiederverwendung und Umwandlung in eine Produktlinie im Rahmen einer Reverse-Engineering existieren bzw. lassen sich effektiv und überdies sogar effizient realisieren? 5) Welche Verfahren existieren bzw. müssen entwickelt werden, um eine „optimale“ Variante entsprechend vorgegebener Eigenschaften in endlicher Zeit zu ermitteln, obwohl die Exploration des Variantenraumes ein NP-vollständiges Problem darstellt?

Hinsichtlich bestehender Werkzeuge gibt es zwar Lösung für kleine Teilbereiche bzw. prototypische Ansätze, jedoch sind die beschriebenen Fragestellungen bisher nicht be-

antwortet. Im Rahmen der Ausarbeitung sind im Speziellen die Fragen zur Durchgängigkeit und zur Unterstützung der User anzugehen, da diese die Akzeptanz des Variantenmanagements und der Wiederverwendung deutlich steigern.

5. Zusammenfassung und Ausblick

In diesem Beitrag wurden die Anforderungen der Branchen Automatisierung, Automotive und Avionik und zugehörige Forschungsfragestellungen vorgestellt, die sich im Zuge einer Erweiterung des SPES 2020 Modeling Frameworks, speziell im Hinblick auf eine Unterstützung des durchgängigen Variantenmanagement und der Wiederverwendung, ergeben. Zukünftig werden nun die einzelnen Forschungsfragestellungen weiter spezialisiert und zugehörige Konzepte, Techniken, Methoden und Werkzeuge entwickelt, die sich bruchfrei integrieren und dann gemeinsam eine umfassende Unterstützung für das durchgängige Variantenmanagement und die Wiederverwendung im Engineering von softwareintensiven eingebetteten Systemen ermöglichen. Die entwickelten Lösungen werden in gemeinsamen Demonstratoren erprobt und ggf. zusätzlich innerhalb der Branchen anhand von Beispielen realer Komplexität hinsichtlich der Skalierbarkeit evaluiert.

Literaturverzeichnis

[AAF03] Alznauer, R.; Auer, K.; Fay, A.: Wiederverwendung von Automatisierungs-Informationen und -Lösungen. Automatisierungstechnische Praxis, Heft 3/2003.

[Br+12] Broy, M.; Damm, W.; Henkler, S.; Pohl, K.; Vogelsang, A.; Weyer, T.: Introduction to the SPES Modeling Framework. In (Pohl, K.; Hönninger H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems, Springer, Berlin, 2012.

[DTW12] Daun, M.; Tenbergen, B.; Weyer, T.: Requirements Viewpoint. In (Pohl, K.; Hönninger H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems, Springer, Berlin, 2012.

[EMV12] Eder, S.; Mund, J.; Vogelsang, A.: Logical Viewpoint. In (Pohl, K.; Hönninger H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems, Springer, Berlin, 2012.

[HK+12] Hilbrich, R.; van Kampenhout, J.R.; Daun, M.; Weyer, T.; Sojer, D.: Modeling Quality Aspects: Real-Time. In (Pohl, K.; Hönninger H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems, Springer, Berlin, 2012.

[H+12] Höfig, K.; Trapp, M.; Zimmer, B.; Liggesmeyer, P.: Modeling Quality Aspects: Safety. In (Pohl, K.; Hönninger H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems, Springer, Berlin, 2012.

[IE00] IEEE Recommended Practice for Architectural Description of Software Intensive Systems. IEEE Standard 1471-2000, 2000.

[IE11] ISO/IEC/IEEE Systems and Software Engineering – Architecture description. ISO/IEC/IEEE Standard 42010:2011, 2011.

[V+12] Vogelsang, A.; Eder, S.; Feilkas, M.; Ratiu, D.: Functional Viewpoint. In (Pohl, K.; Hönninger H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems, Springer, Berlin, 2012.

[W+12] Weber, R.; Reinkemeier, P.; Henkler, S.; Stierand, I.: Technical Viewpoint. In (Pohl, K.; Hönninger H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems, Springer, Berlin, 2012.

Engineering von „Mechatronik und Software“ in automatisierten Anlagen: Anforderungen und Stand der Technik

Thomas Holm, Sebastian Schröck, Alexander Fay

Institut für Automatisierungstechnik
Helmut-Schmidt-Universität / Universität der Bundeswehr Hamburg
Holstenhofweg 85
22043 Hamburg
thomas.holm@hsu-hh.de
sebastian.schroeck@hsu-hh.de
alexander.fay@hsu-hh.de

Tobias Jäger, Ulrich Löwen

Siemens AG CT RTC SYE
San-Carlos-Str. 7
91058 Erlangen
jaeger.tobias@siemens.com
ulrich.loewen@siemens.com

Abstrakt: Dieser Beitrag stellt das Vorgehen und die genutzten Modelle bei der Erstellung von Automatisierungssoftware beim Engineering automatisierter Anlagen dar. Darauf aufbauend wird die Idee, welche der Kontextmodellierung im allgemeinen Software-Engineering zu Grunde liegt, auf Anwendbarkeit innerhalb der Automatisierungstechnik geprüft. Daraus werden Chancen und Herausforderungen abgeleitet werden, welche im Schlussteil dieses Beitrags angeführt sind.

1. Ausgangssituation und Motivation

Das Engineering automatisierter Anlagen ist ein Prozess, indem eine Vielzahl verschiedener Fachdisziplinen integriert werden müssen. So müssen u.a. die Fähigkeiten von Verfahrenstechnik, Mechanik, Elektrotechnik und Automatisierungstechnik genutzt werden, um das Ziel einer funktionsfähigen Anlage zu realisieren. All diese Fachdisziplinen verwenden Software-basierte Werkzeuge, um ihre Planungsaufgaben zu lösen. Diese Werkzeuge werden hier jedoch nicht betrachtet. Die Entwicklung der eigentlichen Anwendungs-Software, d.h. der Software, die in der fertigen Anlage Funktionen erfüllt, erfolgt dabei in der Fachdisziplin Automatisierungstechnik (AT). Diese ist somit für das

Software-Engineering zuständig. Die Automatisierungssoftware (AT-Software) ist projektspezifisch und baut auf den Lösungen der zeitlich vorgelagerten Fachdisziplinen auf. Die Entwicklung der Automatisierungssoftware stellt somit eine der letzten Tätigkeiten im Ablauf des Gesamtprojekts dar. Planungsfehler in den vorgelagerten Fachdisziplinen sowie Inkonsistenzen zwischen den disziplinbezogenen Planungsergebnissen zeigen sich meist erst nach dem Test und der Implementierung der AT-Software. Späte Korrekturen ziehen Nachbesserungen mit zusätzlichen Planungsiterationen nach sich – eine Verlängerung der Projektlaufzeit und ein damit verbundener finanzieller Mehraufwand sind die Folge. Durch Analyse und Beherrschung der Abhängigkeiten zwischen den beteiligten Fachdisziplinen können diese Fehler minimiert werden. Daher stellt dies ein wichtiges aktuelles Forschungsfeld der Automatisierungstechnik dar.

Die zu entwickelnden Anlagen haben sich in den letzten Jahren immer mehr zu mechatronische integrierten Systemen entwickelt [WL11]. Ansätze zur Nutzung mechatronischer Sichtweisen haben sich im Anlagenbau allerdings bisher erst dort durchgesetzt, wo die Systemgrenzen der betrachteten Anlagenteile (= Systembestandteile) in den beteiligten Fachdisziplinen deckungsgleich sind. Im Engineering automatisierter Anlagen wird dies durch die räumliche Verteilung des Systems zusätzlich erschwert. So ergeben sich nicht nur Abhängigkeiten aus der Arbeitsteilung der Fachdisziplinen, sondern auch solche, die aus der verteilten Anordnung der mechatronischen Systembestandteile entstehen.

In diesem Beitrag werden zunächst das Vorgehen beim Engineering automatisierter Anlagen beschrieben und die dabei erarbeiteten Artefakte erläutert. Daran schließt eine Darstellung des charakteristischen Vorgehens bei der Entwicklung von AT-Software an. Der zweite Teil des Beitrags befasst sich mit der Kontextbeschreibung und zeigt, wie diese in der Domäne Automatisierungstechnik angewandt wird. Die Formulierung der sich daraus ableitenden Herausforderungen für die Entwicklung von AT-Software bildet den Abschluss dieses Beitrags.

2. Vorgehen beim Engineering automatisierter Anlagen

Beim Engineering automatisierter Anlagen der Prozess- und Fertigungsindustrie ist eine Vielzahl verschiedener Fachdisziplinen beteiligt. Diese unterscheiden sich in ihren Zuständigkeiten. Typische Fachdisziplinen sind die AT, die Elektrotechnik und die Mechanik. Hinzukommen weitere Fachdisziplinen, die für den Anwendungsbereich der Anlage spezifisch sind. Im Bereich der verfahrenstechnischen Anlagen und dem damit typischerweise verbundenen Transport von fluiden oder granularen Stoffen kommt z.B. der Rohrleitungsbau hinzu. Alle Fachdisziplinen nutzen unterschiedliche, ihren Aufgaben entsprechende Sichten auf das System und verwenden unterschiedliche Software-Werkzeuge. Die Software der Anlage, die das Verhalten dieser bestimmt und in den Speicherprogrammierbaren Steuerungen (SPS) verarbeitet wird, wird jedoch ausschließlich von der AT entwickelt.

Gekennzeichnet sind die zu entwickelten Anlagen durch ihre Einmaligkeit, da sie im Auftrag eines Kunden erstellt und nachfolgend im Rahmen eines Projektes geplant und

realisiert werden [Fa09]. Diese Einmaligkeit entsteht durch die in den Fachdisziplinen getroffenen Entscheidungen und die jeweils erarbeiteten Lösungen. Die Lösungsfindung ist dabei als ein arbeitsteiliger Prozess zu verstehen, bei dem Ergebnisse aus zeitlich vorgelagerten Fachdisziplinen genutzt und um weitere Ergebnisse angereichert werden [Ja11]. Es kann somit von einem Prozess der Informationsanreicherung gesprochen werden.

Die Anlage wird dabei meist top-down vom Grobentwurf sukzessive bis ins Detail geplant, bevor sie realisiert wird. Auf Grund des zeitlichen Drucks während der Projektierung einer Anlage arbeiten die Fachdisziplinen parallel, um die Projektdauer zu verringern. Wo jedoch inhaltlich auf den Ergebnissen anderer aufgebaut wird, werden die Arbeiten sequentiell durchgeführt [Fa09]. Die Ablaufreihenfolge ist somit sowohl innerhalb als auch zwischen den Tätigkeiten nicht beliebig [Ja11]. Die Realisierung der dis-

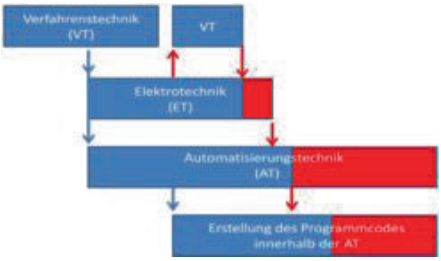


Abb. 1: Zusätzlicher Planungsaufwand durch Iteration und geringe Reifegrade von Planungsergebnissen.

ziplinspezifischen Aufgabe wird durch die planenden Ingenieure mit den ihnen zur Verfügung gestellten Mitteln gelöst. Die Schwierigkeiten ergeben sich meist durch die außerhalb dieses disziplinspezifischen Prozesses entstehenden Randbedingungen. Diese Schwierigkeiten werden durch die Forderung nach einer weiteren Verkürzung der Projektlaufzeit verstärkt. Die Planungstätigkeiten innerhalb der Fachdisziplinen müssen aus diesem Grund meist auf vorläufigen und teilweise lückenhaften Ergebnissen anderer Fachdisziplinen aufgebaut werden. Nach Eintreffen ergänzender oder korrigierender Informationen sind die Fachdisziplinen gezwungen, ihre eigenen Planungsergebnisse iterativ zu überarbeiten [Kl02]. Durch diesen zusätzlichen Planungsaufwand können allerdings Projektverzögerungen entstehen, wie sie in Abb. 1 zu erkennen sind. Die Fachdisziplinen stehen somit während des Engineerings nicht nur durch ihr zeitliches Vorgehen zueinander in einem Abhängigkeitsverhältnis, sondern aufgrund der geplanten Weitergabe der Planungsergebnisse auch in kausalen Abhängigkeitsverhältnissen.

Die Ergebnisse der Planungsphasen werden in Modellen festgehalten. Diese sind meist traditionell gewachsen. Da jede Fachdisziplin unterschiedliche Sichten auf das System hat und auf die spezifischen Aufgabenstellungen ausgerichtete Werkzeuge nutzt, entstanden in den einzelnen Fachdisziplinen verschiedene Modelle und Beschreibungsmittel. So betrachtet die Verfahrenstechnik die Gesamtanlage eher aus einer prozessualen Sicht und legt den Schwerpunkt auf das Zusammenspiel der dafür benötigten prozesstechnischen Geräte und Apparate. Folglich stehen die physikalischen Größen (Durchfluss, Druck, Temperatur) sowie deren Zusammenwirken im Fokus. Die AT hingegen

betrachtet die Anlage als Zusammenwirken von Funktionen, die durch Quellcode innerhalb der SPSen determiniert werden. Die Darstellung innerhalb der AT erfolgt somit durch Modelle, welche die Informationen funktionsorientiert zusammenfassen. Durch diese Fülle an unterschiedlichen Modellen kommt es bei der Informationsübergabe zwischen den Fachdisziplinen immer wieder zu Problemen [KR12].

Gerade bei der Entwicklung umfangreicher automatisierter Anlagen kann die Anzahl der ausführenden Firmen und Zulieferer beträchtlich sein. Dabei können die Planung und Realisierung ganzer Anlagenteile durch externe Zulieferer übernommen werden. Dies kann zu einer Verschärfung der Problematik der Kommunikation an den Informationsschnittstellen führen [WL11].

3. Artefakte im Engineering automatisierter Anlagen

Die zuvor aufgezeigte Beteiligung vieler verschiedener Fachdisziplinen im Engineering automatisierter Anlagen spiegelt sich auch darin wieder, dass eine Vielzahl an Beschreibungsmitteln und Modellen verwendet werden. Die Existenz der unterschiedlichen Modelle und Beschreibungsmittel erschwert einen durchgängigen Planungsablauf. An den Informationsschnittstellen zwischen den Fachdisziplinen kommt es somit, bedingt durch die informationstechnische und semantische Vielfalt, zu Informationsverlust und Fehlinterpretationen von Planungsdaten [Ul09]. Im Folgenden werden die Planungs-Artefakte erläutert, die einen maßgeblichen Einfluss auf die Erstellung der AT-Software haben.

Lastenheft nach [VDI3694]

Den Beginn des Engineerings des AT-Systems stellt die Anforderungserhebung dar. Die Anforderungen werden vom Auftraggeber typischerweise im Lastenheft formuliert. Eine mögliche Struktur wird in der [VDI3694] vorgeschlagen. Ziel ist es, mit dem Lastenheft die Funktion der automatisierten Anlage zu spezifizieren. Dies umfasst sowohl eine Beschreibung des Ist-Zustandes als auch eine exakte Definition der Aufgaben der Automatisierungstechnik sowie deren Rahmenbedingungen. Hierzu zählen auch Anforderungen bzgl. des Projektmanagements, der Qualität und weiterer organisationsspezifischer Charakteristika.

Ein Lastenheft wird vom Auftraggeber meist in Volltext formuliert. Gerade im Bereich der AT werden allerdings zahlreiche Anhänge in Form von Stellenplänen angegeben. Diese umfassen Informationen von vorgegebenen zu nutzenden Komponenten z.B. den Regelkreis einer Füllstandregelung mit Sensor, Datenübertragung, Datenverarbeitung und Aktor.

R&I Fließbild

Das Rohrleitung- und Instrumentierungsfließbild (kurz: R&I-Fließbild) stellt das zentrale Dokument für das Engineering des AT-Systems dar. Es entsteht an der Schnittstelle zwischen Verfahrenstechnik und AT. Ein Ausschnitt eines R&I-Fließbild ist exemplarisch in Abb. 2 dargestellt. Es wird durch Informationsanreicherung auf Grundlage des

Verfahrensfließbild erstellt. Im R&I-Fließbild sind Informationen über Stoffflüsse sowie wichtige Informationsflüsse enthalten. Primär werden die Rohrleitungen, welche die verschiedenen am Prozess beteiligten Apparate miteinander verbinden, dargestellt. Darüber hinaus ist hier definiert, wo welcher Messwert aufgenommen werden soll. Durch die Nutzung von Akronymen wird definiert, wo und wie dieser Messwert verarbeitet wird. Die Zusammenstellung aus Sensor und Benennung der Aufgabe (nach [DIN62424] PCE-Aufgabe, Process Control Engineering) wird auch als Messstelle bezeichnet.

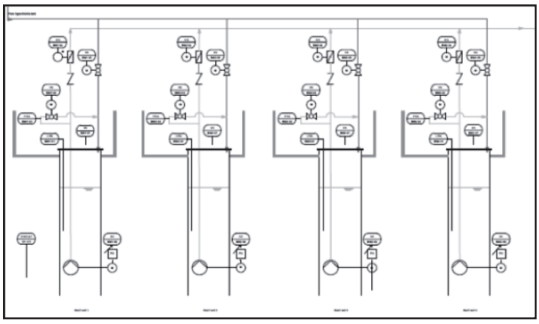


Abb. 2: Auszug aus einem R&I-Fließbildes einer Meerwasserentsalzungsanlage

Stellenblatt, Stellenplan und Stellenverzeichnis

Aufbauend auf den Informationen im R&I-Fließbild werden von der AT weitere prozessleittechnische Artefakte erstellt.

Das PLT[1]-Stellenverzeichnis stellt in geordneter und übersichtlicher Form die wesentlichen Informationen zu den im R&I-Fließbild enthaltenen PCE-Aufgaben dar. Die eindeutige PCE-Aufgabe sowie die PCE-Kategoriebezeichnung [DIN62424] dienen der Funktionsbeschreibung und Kurzbeschreibung der PCE-Aufgabe.

Das PLT-Stellenverzeichnis wird durch die jeweiligen PLT-Stellenblätter konkretisiert, wobei zu jedem Eintrag des PLT-Stellenverzeichnisses ein PLT-Stellenblatt entsteht.

Das PLT-Stellenblatt (Abb. 3 links) gibt Aufschluss über die zu einer PCE-Aufgabe gehörenden Informationen. Diese umfassen die allgemeine Kennzeichnung der Aufgabe, die verfahrenstechnischen Betriebsdaten (Bezeichnung, Zusammensetzung, Stoffeigenschaften), den genauem Einbauort und die für die PCE-Aufgabe genutzten Geräte. Die graphische Repräsentation des PLT-Stellenblatts ist der PLT-Stellenplan (Abb. 3 rechts). Er gibt die zur PCE-Aufgabe gehörenden Einrichtungen, Antriebe, Stellglieder, Signalgeber, Befehlsgeräte sowie alle zugehörigen Funktionselemente, ihre örtliche Lage und ihre Verbindungen untereinander an [Fe01]. Im PLT-Stellenplan werden die Funkti-

[1] In der [DIN62424] wird lediglich die Nomenklatur PCE (Process Control Engineering) verwendet, die entsprechenden Dokumente jedoch nicht explizit benannt. In der gängigen Literatur (bspw. [Fe01]) wird hingegen in Bezug auf die Dokumente das Akronym PLT (Prozessleittechnik) verwendet. Eine Anpassung dieser Namensgebung fand bisher nicht statt. Die PLT ist als Teil der AT anzusehen.

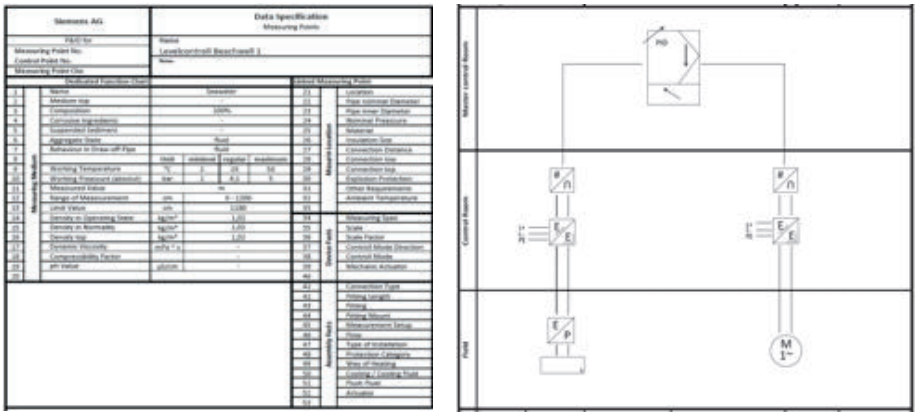


Abb. 3: Auszug eines PLT-Stellenblatts & Auszug eines PLT-Stellenplans

onselemente mit Sinnbildern nach [DIN19227][2] dargestellt. Der PLT-Stellenplan stellt die enge Verknüpfung der PCE-Aufgabe mit der Realisierung durch Softwarebausteine im Prozessleitsystem dar. Klappt man den PLT-Stellenplan gedanklich in der Mitte vertikal auf und ersetzt die Funktionselemente durch entsprechende Funktionsbausteine, erhält man einen Continuous Function Chart (kurz: CFC).

Continuous Function Chart (CFC) und Sequence Function Chart (SFC)

Continuous Function Chart (CFC) und Sequence Function Chart (SFC) werden typischerweise von der AT erstellt und sind nach einer Kompilierung auf den SPS der Anlage lauffähig.

CFC ist eine graphische Programmiersprache für SPS. Obwohl diese Darstellung keiner der in der [IEC61131] genormten Sprachen entspricht, können CFCs als Quasi-Standard angesehen werden. In CFCs werden Funktionsbausteine logisch miteinander verschaltet und so eine Software-Funktion realisiert. Ein CFC-Plan kann mehrere Teilpläne enthalten, wobei jeder Teilplan in zwei Spalten zu je drei Blättern aufgeteilt ist. Ein Blatt verfügt jeweils an der linken und rechten Seite über Plananschlüsse für die Eingangs- und Ausgangsvariablen. Über diese ist es möglich, verschiedene Blätter, Teilpläne oder Pläne miteinander zu verschalten. Auch können mit Hilfe von CFC-Plänen hierarchische Strukturen realisiert werden. Es ist also möglich, einen CFC-Plan innerhalb eines anderen CFC-Plans zu erstellen.

SFC ist eine graphische Programmiersprache, die in der [IEC61131] spezifiziert ist. Mit Hilfe von Transitionen und Schritten kann in SFC ein ablauforientiertes Verhalten eines Systems programmiert werden. Die jeweiligen Schritte beinhalten meist eine Ausführungsbestimmung für vordefinierte Aktionen. Die Transitionen zwischen den Schritten sind meist an Bedingungen geknüpft und steuern so den Übergang zwischen den einzel-

[2] Die DIN 19227 wurde zurückgezogen. DIN 19227-1 ist bis auf Unterabschnitt 3.9 in der DIN EN 62424 enthalten. DIN19227-1 Unterabschnitt 3.9 soll zukünftig in die ISO 10628 aufgenommen werden.
DIN 19227-2 wird voraussichtlich in DIN EN 60617 eingebracht werden.

nen Schritten. Schritte und Transitionen sind mit gerichteten Kanten verbunden. In der einfachsten Form verbindet eine Transition einen Schritt bzw. umgekehrt. Komplexere Sequenzen können auch mit Sprüngen innerhalb des SFC-Plans oder Alternativ- und Parallelzweigen modelliert werden.

Weitere Artefakte

Ergänzend zu den zuvor angeführten Artefakten werden im Engineering der AT von Anlagen auch weitere verwendet, die hier nur in Kürze angeführt werden. Das Human-Machine-Interface (kurz HMI) stellt die Gesamtheit der Bedienbilder sowie der hinterlegten Software dar, die eine Bedienung der Anlage überhaupt erst ermöglichen. Ebenso unerlässlich ist die Erstellung der Hardware-Konfiguration. Hier wird in der Software definiert, welches Automatisierungsgerät verwendet wird. Desweiteren werden die Parameter der Kommunikations-Busse und die Adressenräume und deren Speicherplatzzuweisung innerhalb der SPS festgelegt.

4. Der Kontext für das Engineering automatisierter Anlagen

Die Kontextmodellierung ist im Bereich der allgemeinen Softwareentwicklung weit verbreitet. Nach [PR11] ist der Systemkontext (nachfolgend kurz: Kontext) derjenige Teil der Umgebung, der für die Anforderungsgewinnung aber auch deren Verständnis relevant ist. Im Gegensatz zur allgemeinen Softwareentwicklung wird im Engineering automatisierter Anlagen der Kontext i.d.R. nicht explizit modelliert. Dies stellt einen entscheidenden Unterschied zwischen den verglichenen Domänen dar. Der Kontext ist jedoch implizit in verschiedenen Artefakten enthalten, wobei die Lage der Systemgrenze Auswirkungen darauf hat, welche der zuvor genannten Artefakte dem System und welche dem Kontext zuzuordnen sind. Um diesen Sachverhalt klären zu können, bedarf es der Erläuterung des Systems bzw. dessen Interaktion mit dem Kontext.

4.1 System und Kontext aus der Sicht des Anlagenbaus

Im Anlagenbau umfasst das zu entwickelnde System i.d.R. die gesamte Anlage mitsamt den verfahrenstechnischen oder fertigungstechnischen Prozessen. Hier ist das relevante System die gesamte Anlage mit Prozess, allen Apparaten, Rohrleitungen, Gebäuden, Kommunikationssystemen, sowie der nötigen Informationsverarbeitung.

Der Kontext dieses sehr weit gefassten und damit stark interdisziplinären Systems ist durch verschiedene Rahmenbedingungen geprägt. Die Anlage wird in eine Umgebung integriert die spezifische Charakteristika aufweist. Sind diese für das Engineering bzw. den Betrieb der Anlage relevant, zählen diese zum Kontext. Hinzu kommen aber auch behördliche Vorgaben, die es zu beachten gilt. Wie in Abb. 4 ersichtlich, beinhaltet der Kontext Aspekte, die alle Projektphasen des Engineerings und den gesamten Lebenszyklus der Anlage beeinflussen.

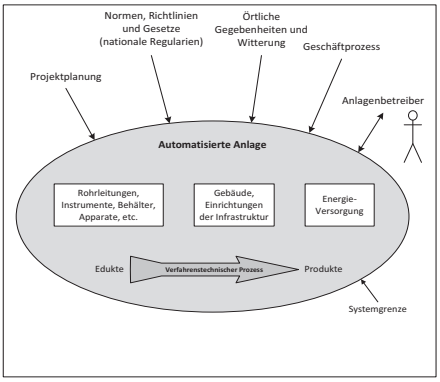


Abb.4: Kontext beim Engineering der Anlage

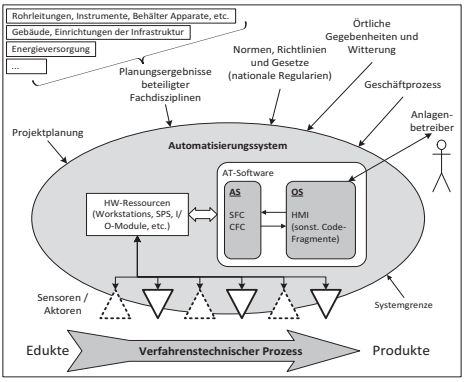


Abb. 5: Kontext beim Engineering des AT-Systems

Diese umfassen sowohl Informationen zu vorherrschenden klimatischen Bedingungen aber auch Forderungen, die den Geschäftsprozess betreffen. So werden u.a. Komponenten zugekauft, die zuvor von einem externen Zulieferer konzipiert wurden. Diese können in ihren Eigenschaften jedoch nicht mehr angepasst sondern maximal parametrisiert werden und sind somit ebenfalls als Randbedingung anzusehen. Wird die Systemgrenze verschoben und auf das AT-System fokussiert (Abb. 5), führt dies zu einer Veränderung des Kontexts. Da sich das AT-System innerhalb der Anlage befindet, enthält der Kontext nun zusätzlich auch von anderen Fachdisziplinen geplante und realisierte Teile der Anlage, die für das AT-System von Relevanz sind.

4.2 System und Kontext aus der Sicht der Automatisierungstechnik

Richtet man den Fokus auf das von der AT zu entwickelnde System, verändert sich der Kontext. Lag die Entwicklung des verfahrenstechnischen Prozesses (formalisiert dargestellt im R&I-Fließbild) im Schwerpunkt der Aktivitäten der Verfahrenstechnik, wird dieser nun zum Kontext. Das Planungsergebnis der Verfahrenstechnik (R&I-Fließbild) ist daher Anforderungsdokument der AT. Darauf aufbauend werden das AT-System entwickelt und die Planungsergebnisse in den oben beschriebenen Planungsartefakten modelliert.

Die Wechselwirkung mit dem Prozess geschieht über entsprechende Hardware (Sensoren/Aktoren), die jedoch auch aus dem AT-System heraus konfiguriert wird somit nicht in Gänze zum Kontext zuzurechnen ist. Einige Teile der Hardware werden allerdings auch im AT-System zugekauft. Die durch den Zukauf festgelegten Eigenschaften stellen somit Randbedingungen dar. Wie in Abb. 5 ersichtlich, hat die AT-Software über das HMI eine Schnittstelle zum Anlagenfahrer – die diesbezüglich formulierten Anforderungen sind ebenfalls in Teilen dem Kontext zuzuschreiben. Planungsergebnisse anderer Disziplinen stellen ebenfalls Randbedingungen dar und können folglich direkte Auswirkungen auf das AT-System haben, weshalb auch diese hier als Teil des Kontexts anzusehen sind.

Einige Einflussfaktoren, die den Kontext der gesamten Anlage betreffen, wirken auch auf das AT-System. Diese sind i.d.R. im Lastenheft der Anlage dokumentiert, da so das Wissen des Anlagebetreibers, z.B. über die nationalen Gegebenheiten, weiter gegeben werden kann und sollte.

In diesem Spannungsfeld aus prozessualen, technischen, gesetzlichen, klimatischen und menschlichen Einflüssen muss das AT-System bestehen, was eine erhebliche Herausforderung darstellt, da in der Regel keine Anlage der anderen gleicht. Somit steht auch jedes AT-System einem, wenn auch nur in Teilen, abweichenden Kontext gegenüber.

4.3 Abgrenzung des Kontextes zu anderen Domänen

Wie zuvor erwähnt, wird der Kontext in anderen Domänen z.B. der allgemeinen Softwareentwicklung im Detail modelliert, um die Anforderungen zu erheben, zu dokumentieren und möglichst auch zu formalisieren, um dadurch auch das Systemverhalten validieren zu können. In Domänen mit großen Zahlen gleichartiger Produkte ist es möglich, die durch den Modellierungsaufwand anfallenden Mehrkosten auf eine große Stückzahl von verkauften Produkten umzulegen. Da jedoch im Anlagenbau der Großteil der Anlagen einmalig ist, schlägt der Aufwand für die Modellierung des Kontextes in voller Höhe bei den Projektkosten zu Buche. Aus diesem Grund wird dieser nicht explizit modelliert. Um die Ermittlung und Modellierung des relevanten Kontextes dennoch zu ermöglichen, bedarf es der Verwendung effizienter Methoden, die die Ermittlung und Nutzung des Kontexts unterstützen. Bedingt durch den stark arbeitsteiligen Entwicklungsprozess im Anlagenbau müssten diese Methoden in der Lage sein, den Kontext je nach Lage der Systemgrenzen anzupassen bzw. aus vorhandenen Planungsartefakten abzuleiten. Dies bedarf jedoch des detaillierten Wissens der Wirkbeziehungen zwischen den Planungsergebnissen der verschiedenen Fachdisziplinen. Anstrengungen bzgl. einer methodengestützten Kontextmodellierung und -analyse können erst der Analyse dieser Wirkbeziehungen nachgelagert stattfinden und rücken zunächst in den Hintergrund.

5. Software für automatisierte Anlagen

Die Entwicklung der AT-Software ist eine der letzten Arbeitsschritte während des Engineerings automatisierter Anlagen. Es baut auf den getroffenen Auslegungsentscheidungen zeitlich vorgelagerter Fachdisziplinen auf. Im Folgenden werden die Auswirkungen dessen auf die AT-Software einer Anlage beleuchtet.

5.1 Architektur der Automatisierungssoftware

Der Aufbau und die Entwicklung von automatisierungstechnischer Software, also der Realisierung von Steuerungs- und Regelungsabläufen, unterscheiden sich von der einer allgemeinen Software. Bei der AT-Software ist die Kopplung zum technischen System eine sehr viel engere, als dies bei allgemeiner Software der Fall ist. Bei AT-Software ist der technologische Prozess, der die verfahrens- oder fertigungstechnischen Vorgänge

beschreibt, strukturgebender Faktor. Der technologische Prozess ist somit die zentrale Information, die im Kontext des AT-Systems dargestellt werden sollte. Der Betrieb einer automatisierten Anlage ist meist auf eine Vielzahl verschiedener Softwaresysteme aufgeteilt, die verschiedenen logischen Ebenen zugeteilt werden können. Im Folgenden wird lediglich Bezug genommen auf die Ebene des Prozessleitsystems, da hier das automatisierungstechnische Know-How einfließt (vgl. [St12]).

In der Prozessindustrie ist das Prozessleitsystem typischerweise in zwei Teile unterteilt. Die Prozessführung erfolgt über eine oder mehrere „Operator Station" (OS). Diese realisieren das Verhalten und die Auswertung der HMIs, über die die Anlage bedient und beobachtet wird. Die Steuerungs- und Regelungsfunktionen werden auf einer oder mehrerer „Automation Station" (AS) ausgeführt. Diese bestehen aus den zuvor erwähnten SPSen und müssen auch bei Ausfall der OS rückwirkungsfrei arbeiten[3].

Die AS ist i.d.R. anhand der Anlagentopologie strukturiert. Somit richtet sich die Struktur der Software im besten Fall nach der realen Struktur der Anlage. Die Softwarefunktionen der Anlagenelemente sind in den entsprechend CFC-Plänen durch Funktionsbausteine und deren Verschaltung repräsentiert. Hier ist das Verhalten einer jeden Komponente abgelegt, in Summe ist hier somit auch das Anlagenverhalten definiert.

Dem gegenüber steht die OS, also der Teil der AT-Software, der die Bedienbilder, also HMIs beinhaltet. Die zum Verfahren der Anlage nötigen Bedienoberflächen sind i.d.R. hierarchisch aufgebaut. Einem grobstrukturiertem Anlagenbild, welches meist eine Übersicht über die wichtigsten Anlagenparameter aufzeigt, sind detailliertere Bedienbilder untergeordnet. Hier kommen rein grafische Elemente sowie Elemente mit weiteren Softwarefragmenten zum Einsatz, die mit den Variablen des AS verknüpft sind. Dies ist umso einfacher, je mehr die Struktur der Bedienbilder mit der der realen Anlage und damit der der AS übereinstimmt. Dies ist jedoch nicht zwingend erforderlich und auch nicht immer von Vorteil, da Anforderungen an die Usability dem entgegenstehen können. Die Informationen bezüglich Struktur, aber auch Art und Gestalt der HMI sind Informationen, die einem Kontext zu entnehmen wären.

5.2 Vorgehen bei der Erstellung der Automatisierungssoftware

Bei der Erstellung der AT-Software wird entsprechend der zuvor erläuterten Trennung zwischen AS und OS vorgegangen. Die AS wird unter der Verwendung von Typicals (die man sich als Kopiervorlagen vorstellen kann) erstellt. So lassen sich eine Vielzahl von CFC-Plänen für Messstellen mit bekannten Funktionen und Komponenten ableiten. Die relevanten Funktionsbausteine und Variablen werden aus spezifischen Bibliotheken instanziiert. Der Prozessleittechniker muss diese korrekt verknüpfen sowie SFC-Pläne erstellen, in denen Ablaufsequenzen definiert sind. Die Strukturierung und Hierarchisierung des Programmcodes der AS erfolgt typischerweise entlang der funktionalen Anlagenstruktur. Da diese zum Zeitpunkt der Software-Erstellung bereits vorliegt, ist die Architektur der AT-Software vorgegeben. Dies ist ein entscheidender Unterschied zur Entwicklung einer Software ohne starke Bindung zu einem technischen System.

[3] Die Benennung entstammt dem Prozessleitsystem PCS7 der Siemens AG und wird im Weiteren verwendet.

Die Erstellung der Bedienbilder sowie deren Strukturierung und Programmierung erfolgt im Engineeringwerkzeug der OS. Dieser Schritt erfolgt in Teilen automatisch und unter Nutzung der von der AS übernommenen Anlagenstruktur. Dabei sind den Funktionsbausteinen in den CFC-Plänen Visualisierungselemente zugeordnet. Wird ein CFC-Plan in einem funktionalen Element der AS-Struktur instanziiert, wird dieses Visualisierungselement dem entsprechenden Bedienbild zugewiesen. Die hierarchische Struktur der Bedienbilder entspricht somit i.d.R. der funktionalen Struktur innerhalb der AS. Die entstandenen Bedienbilder verfügen zu diesem Zeitpunkt somit über Elemente, die automatisch mit den entsprechenden Variablen der Anlagenkomponenten verknüpft sind. Die Elemente auf dem Bedienbild müssen nun durch den Entwickler angeordnet und ggf. mit graphischen Elementen z.B. nicht animierten Rohrleitungen erweitert werden. Die Bedienbilder sollten so den funktionalen Ablauf der Anlage darstellen.

6. Herausforderungen und Chancen

Die Software ist innerhalb der AT von zentraler Bedeutung. Ihre Entwicklung stellt eine der letzten Entwicklungsschritte im Engineering automatisierter Anlagen dar. Hier findet somit die Integration der Lösungen der vorgelagerten Fachdisziplinen statt. Gleichzeitig zeigen sich hier mögliche Fehlplanungen. Um Effizienz und Qualität der Planung und Realisierung der AT und damit der Planung und Erstellung von automatisierten Anlagen insgesamt zu verbessern, sind mehre Ansatzpunkte vorhanden:

- Die Wirkbeziehungen zwischen den einzelnen Fachdisziplinen müssen besser identifiziert, dokumentiert und im Anschluss modelliert werden, um im Falle einer Änderung in einer Fachdisziplin deren Auswirkungen auf die anderen Fachdisziplinen valide abschätzen zu können.
- Neben dem disziplinorientierten Engineering sollte verstärkt das Engineering unter Nutzung mechatronischer Sichtweisen weiter vorangetrieben werden.
- Es gilt, das Engineering der vorgelagerten Fachdisziplinen methodisch zu verbessern, um somit die Fehler und damit die ungeplanten Iterationen zu reduzieren. Dies hat eine bessere Planungsgrundlage für die AT, und damit eine Software höherer Güte, zur Folge.
- Es müssen durchgängige Modelle und Methoden für die Entwicklung von automatisierten Anlagen erarbeitet werden. Noch sind diesbezügliche Ansätze stark abhängig von den verwendeten Engineering-Werkzeugen. Ebenso bestehen oft noch Schwächen bezüglich der methodischen Wiederverwendbarkeit. Hier müssen bisher werkzeugspezifisch existierende Teillösungen etabliert, standardisiert und in die durchgängige Methode integriert werden.
- Eine methodische Identifikation des relevanten Kontexts könnte einen Mehrwert für alle Projektbeteiligten bezüglich Validierung und Optimierung, aber auch im Fall einer Modernisierung der Anlage bieten. Auf Basis der zuvor erläuterten Rahmenbedingungen sind hier jedoch effiziente Methoden zu erarbeiten, um den bei der Modellierung entstehenden Aufwand reduzieren zu können.

Literaturverzeichnis

[KR12] Kühnl, C., Rothhöft, M.: Unzufrieden mit Engineering Schnittstellen. In: Computer-automation.de 06/12, 2012

[DIN10628] DIN EN ISO 10628, März 2001.DIN EN ISO 10628 - Fließschema für verfahrenstechnische Anlagen.

[DIN19227] DIN 19227, Februare 1991.DIN 19227 - Graphische Symbole und Kennbuchstaben für die Prozeßleittechnik, Darstellung von Einzelheiten.

[DIN62424] DIN EN 62424, April 2009.DIN EN 62424 - Festlegung für die Darstellung von Aufgaben der Prozessleittechnik in Fließbildern und für den Datenaustausch zwischen EDV-Werkzeugen zur Fließbilderstellung und CAE-Systemen.

[Fa09] Fay, A.: Effizientes Engineering komplexer Automatisierungssysteme. In: Schnieder, Ständer (Hrsg.): Wird der Verkehr automatisch sicherer?; 04. September 2009 in Braunschweig. Braunschweig: iVA, S. 43–60, 2009.

[Fe01] Felleisen, M.: Prozeßleittechnik für die Verfahrenstechnik, Oldenbourg Industrieverlag, 2001 München.

[IEC61131] DIN IEC 61131-3, Juni.2009.IEC 61131-3 - Speicherprogrammierbare Steuerungen - Teil 3: Programmiersprachen.

[Ja11] Jäger, T., Fay, A., Figalist, H., Wager, T.: Systematische Risikominimierung im Engineering mit Abhängigkeitsanalyse und Schlüsseldokumenten; in: Tagungsband „Automation 2011“;, Baden-Baden, VDI-Verlag, Düsseldorf

[Kl02] Klein, R.A., Anhäuser, F., Burmeister, M., Lamers, J.: Planungswerkzeuge aus Sicht eines Inhouse-Planers. – In: atp-Automatisierungstechnische Praxis 44 (2002) Nr. 1, S. 46-50.

[PR11] K. Pohl, C. Rupp: Basiswissen Requirements Engineering: Aus- und Weiterbildung zum "Certified Professional for Requirements Engineering" ; Foundation Level nach IREB-Standard. 3. Aufl. Heidelberg: dpunkt-Verl, 2011.

[St12] M. Strube, I. Kühl, T. Holm, A. Fay, R. Mühlfeld, H. Figalist: Modellierung von Kommunikationsschnittstellen bestehender Automatisierungslösungen in Modernisierungsprojekten auf Basis von Signallisten. In: Automation 2012: Der 13. Branchentreff der Mess- und Automatisierungstechnik, Vol. 13, S. 95–98, 2012.

[Ul09] A. Ulrich: Entwicklungsmethodik für die Planung verfahrenstechnischer Anlagen. Als Ms. gedr. Düsseldorf: VDI-Verl, 2009 (Fortschritt-Berichte VDI Reihe 20, Rechnerunterstützte Verfahren, 425).

[VDI3694] VDI/VDE 3694, Januar 2008.VDI/VDE 3694 - Lastenheft/Pflichtenheft für den Einsatz von Automatisierungssystemen.

[WL11] T. Wagner, U. Löwen: Modellierung: Grundlage für integriertes Engineering; in: Tagungsband „Automation 2011“;, Baden-Baden VDI-Verlag, Düsseldorf

Herausforderungen an ein durchgängiges Variantenmanagement in Software-Produktlinien und die daraus resultierende Entwicklungsprozessadaption*

Christian Manz
Research and Development
Daimler AG, Germany
christian.c.manz@daimler.com

Manfred Reichert
Institut für Datenbanken und Informationssysteme
Universität Ulm, Germany
manfred.reichert@uni-ulm.de

Abstract: In der Automobilindustrie werden Kundenwünsche zunehmend mittels Elektrik/Elektronik-Komponenten und zugehöriger Software realisiert. Dies führt in der fortlaufenden Entwicklung zu einer steigenden Komplexität und Variabilität der Software. Software-Produktlinien (SPL) beschreiben eine Methodik, um solch variantenreiche Softwaresysteme zu beherrschen. Ein durchgängiges Variantenmanagement betrifft sämtliche Entwicklungsphasen und deren Abstraktionsebenen. So ermöglicht es ein verbessertes Tracing, zielgenaue Change-Impact-Analysen und durchgängige Fehlerbeseitigungen. Um die Anforderungen an ein durchgängiges Variantenmanagement besser zu verstehen, untersuchten wir eine existierende SPL und fanden voneinander isoliert erstellte Merkmalmodelle vor. Als Ursachen hierfür lassen sich u. a. differenzierte Sichtweisen auf Softwareproduktvarianten in der Entwicklung sowie die fehlende Prozessverankerung der Merkmalmodellierung ausmachen. Eine Harmonisierung der isoliert erstellten Merkmalmodelle gestaltet sich aufwändig und erschwert ein durchgängiges Variantenmanagement.

In diesem Beitrag werden industrielle Herausforderungen zum Erreichen eines durchgängigen Variantenmanagement in der Praxis erläutert. Ziel ist es, bestehende Merkmalmodelle zu harmonisieren und Assoziationen zwischen Merkmalen über die gesamten Entwicklungsphasen und den vorhandenen Abstraktionsebenen herzustellen. Für eine standardisierte Merkmalmodellierung wird zudem eine Adaption des Entwicklungsprozesses beschrieben.

1 Einführung

In der Automobilindustrie führen Kundenanforderungen (z.B. Komfortfunktionen, Produktsicherheit, Umweltbelastung) zu einer steigenden Anzahl unterschiedlicher Produktvarianten. Weltweite Produktvermarktungen sowie kontextspezifische Einschränkungen

*Diese Arbeit wurde teilweise im Rahmen des BMBF-Projekts SPES XT (01IS12005) gefördert.

(z.B. länderspezifische Regularien) sind zusätzliche Treiber von Produktvarianten. Üblicherweise wird ein Großteil dieser Produktvarianten in der Automobilindustrie heutzutage mittels Software realisiert. *Software-Produktlinien (SPL)* [PBL05, WL99, TH02] zeichnen sich durch eine geplante und systematische Wiederverwendung von Softwareartefakten aus, die in verschiedenem Kontext (z.B. verschiedene Fahrzeuge, Länder) zum Einsatz kommen. Ihr Ziel ist es, ähnliche Softwareprodukte gemeinsam mit geringeren Kosten, verkürzten Entwicklungszeiten und steigender Qualität zu entwickeln.

Das *Variantenmanagement* in einer SPL erfordert einen anderen Ansatz im Vergleich zur klassischen Entwicklung von Einzel-Softwareprodukten [PBL05]. Die Herausforderung besteht darin, sowohl die Gemeinsamkeiten als auch die Variabilität zwischen Produkten einer SPL zu handhaben. Gemeinsamkeiten sind Eigenschaften, die in allen Produkten vorhanden sind; so besitzt jedes Auto einen Motor. Variabilität hingegen beschreibt die Unterschiede zwischen den Produkten (z.B. Standard, besonders Leistungsstark oder signifikant Ökologisch).

Im Variantenmanagement von SPL bildet die *Merkmalmodellierung* eine viel verwendete Technik, um die Gemeinsamkeiten und Variabilität zu beschreiben. Restriktionen und Relationen zwischen Merkmalen schränken dabei die Variabilität innerhalb eines Merkmalmodells ein [CE00]. Der Begriff *Merkmal* wird in SPL in verschiedener Weise verwendet. In diesem Beitrag beziehen wir uns auf die Definition von Czarnecki [CE00]: „Ein Merkmal (*Feature*) beschreibt eine Eigenschaft eines Konzepts für eine bestimmte Interessensgruppe.“ Die Methode der Merkmalmodellierung wurde ursprünglich als *Feature-Oriented Domain Analysis (FODA)* [Ka90] eingeführt und im Laufe der Zeit im industriellen Einsatz um bestimmte Notationen erweitert [CHE05]. Es stehen daher mehrere Methoden zur Merkmalmodellierung zur Verfügung. Dieser Beitrag fokussiert auf die Assoziationen zwischen Merkmalen und ist von der verwendeten Methode der Merkmalmodellierung unabhängig.

Abbildung 1 illustriert beispielhaft den Zusammenhang zwischen einem Merkmal (s. oberer Teil von Abb. 1) und den Artefakten (s. unterer Teil von Abb. 1). Über die gestrichelten Linien erkennbar, ist die Beziehung zwischen den Merkmalen und den verschiedenen Artefakten im Entwicklungsprozess [PM08]. Der veranschaulichte Entwicklungsprozess dient hierbei nur zur Illustration der Variabilität über einen kompletten Produktlebenszyklus und kann für weitere Entwicklungsprozesse verallgemeinert werden. Textuelle Anforderungen, UML-Diagramme, Funktionsmodelle, Testspezifikationen, Regressionstests oder Applikationsparameter sind Beispiele für Artefakte.

In der von uns untersuchten SPL findet sich beispielsweise ein Merkmalmodell, das von den Softwareanforderungen abgeleitet wurde und mit verschiedenen Artefakten (z.B. Funktionsmodelle) in Verbindung steht. Gleichzeitig finden wir in der untersuchten SPL weitere Merkmalmodelle (z.B. Applikation), die wiederum mit ihren Artefakten in Verbindung stehen. Durch Harmonisierung der bestehenden Merkmalmodelle streben wir insbesondere ein durchgängiges Variantenmanagement an. Eine Harmonisierung von isoliert voneinander erstellten Merkmalmodellen gestaltet sich auf Grund verschiedener Sichtweisen auf Softwarevarianten sehr schwierig. So finden sich in der untersuchten SPL nicht alle Varianten der Applikation (Variation durch Post-Build Parametrierung) in den dokumentierten Anforderungen wieder. Beispielsweise wurden mehrere Merkmale der Anforderungen

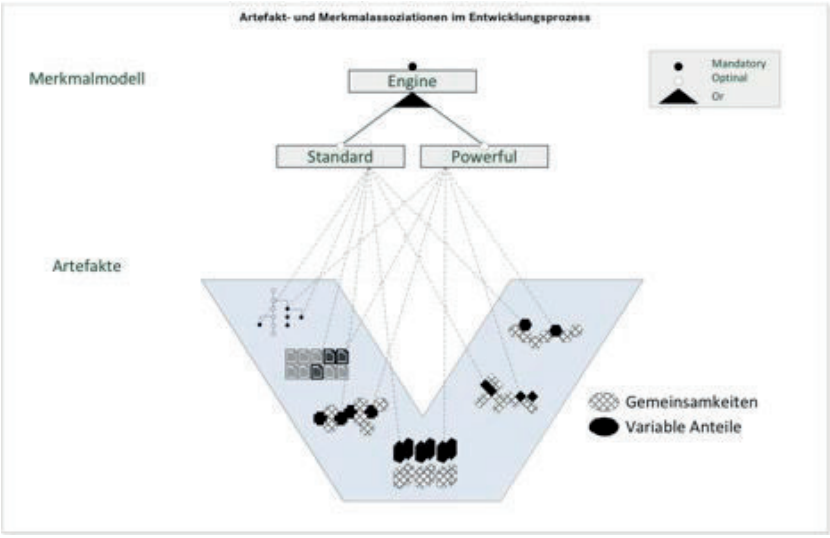


Abbildung 1: Artefakt und Merkmalassoziationen im Entwicklungsprozess

mittels eigener Merkmale der Applikation gruppiert. Dies führt durch die Definition neuer Merkmale unvermeidbar zu verschiedenen Merkmalmodellen, welche im Nachhinein schwer zu harmonisieren sind.

In dieser Arbeit werden Operatoren zur Harmonisierung isolierter Merkmalmodelle beschrieben. Des Weiteren zeigen wir, dass die Problemstellung der isolierten Merkmalmodellierung auf eine unzureichende Verankerung der Merkmalmodellierung im Entwicklungsprozess zurückzuführen ist. Daraus leiten wir die notwendige Adaption des Entwicklungsprozesses (Prozessadaption) ab. Kapitel 2 beschreibt unsere Vorstellung von einem durchgängigen Variantenmanagement, die vorliegende Ausgangssituation der untersuchten SPL und daraus resultierende Herausforderungen in der Praxis. In Kapitel 3 werden Operatoren bei isolierten Merkmalmodellen für ein durchgängiges Variantenmanagement beschrieben sowie notwendige Prozessadaptionen skizziert. Verwandte Arbeiten und eine Zusammenfassung werden in den Kapiteln 4 und 5 erläutert.

2 Durchgängiges Variantenmanagement

Das Variantenmanagement einer SPL sollte sich auf den gesamten Entwicklungsprozess beziehen. Softwarevarianten treten dabei in verschiedenen Entwicklungsphasen und zugehörigen Abstraktionsebenen auf. Dementsprechend finden sie sich in verschiedenen Artefakten wieder. Typische Phasen der automobilen Softwareentwicklung sind Anforderungsanalyse, Design, Implementierung, Integration, Test und Applikation. Des Weiteren kön-

nen verschiedene Abstraktionsebenen (z.B. System, Sub-System oder Sub-Sub-System) in den einzelnen Phasen beobachtet werden. Für ein umfassendes Tracing in einer SPL sind nach [BBM05] folgende Dimensionen relevant: Variabilität, Abstraktion und Entwicklungsphasen. Ziel dieses Beitrages ist die Beschreibung einer Methodik, mittels der sich die isoliert erstellten Merkmalmodelle für ein durchgängiges Variantenmanagement nutzen lassen. Dabei fokussieren wir ausschließlich auf die Beschreibung von Variabilität mittels Merkmalen und deren Beziehungen untereinander.

2.1 Ausgangssituation

Für die Erzielung eines besseren Verständnisses, was durchgängiges Variantenmanagement bedeutet, haben wir eine existierende SPL für Motorsteuerungen bei Daimler Trucks analysiert. Diese SPL wird weltweit von verteilten Teams im Embedded Software Engineering entwickelt und weist eine Vielzahl von Varianten auf, z.B. Zylinderanzahl, Hubraum, Leistung, Drehmoment, Abgasnorm, Markt, Motorplattform. Mehr als tausend Varianten werden dabei in der Applikation realisiert. Ferner konnten wir beobachten, dass die untersuchte SPL zur Verwaltung und Dokumentation voneinander isoliert gepflegte Merkmalmodelle aufweist. Im aktuellen Entwicklungsprozess werden keine Merkmale mit vor- bzw. nachgelagerten Entwicklungsphasen oder Abstraktionsebenen abgestimmt.

Weiterhin haben wir die jeweils Verantwortlichen zu ihrer Sichtweise auf Softwarevarianten und den verwendeten Merkmalen in den jeweiligen Entwicklungsphasen und Abstraktionsebenen befragt. Treiber der Merkmalmodellierung ist dabei die jeweilige Konfiguration und Dokumentation der Varianten. Eine vollständige Beschreibung der Varianten fand sich in keinem der Modelle wieder. Gründe dafür sind unter anderem die Unwissenheit über weitere Varianten oder fehlendes Wissen zur weiteren technischen Realisierung in der Entwicklung.

Unsere Analyse hat das Vorhandensein unterschiedlicher Sichtweisen auf die Variabilität eines Produktes bestätigt. [PM08, Ro11a] beschreiben diese Sichtweisen mit *externer* und *interner Variabilität*. Externe Variabilität umfasst die Umgebung bzw. Konfiguration einer Komponente, und ist üblicherweise in den Anforderungen beschrieben. Interne Variabilität dagegen entsteht typischerweise aus technischen Gründen im Verlauf der Realisierung und wird in den Entwicklungsartefakten dokumentiert. Da in den Anforderungen normalerweise die technische Realisierung (z.B. Architektur der Software, Aufteilung der Funktion auf Steuergeräte kann von der gewählten Sonderausstattung abhängig sein) nicht definiert wird, können im fortlaufenden Entwicklungsprozess neue Merkmale hinzugefügt werden. Ein durchgängiges Variantenmanagement mittels einem Merkmalmodell, welches einzig von den Anforderungen abgeleitet ist, konnte in der untersuchten SPL nicht realisiert werden. In Folge von externer und interner Variabilität und den damit verbundenen differenzierten Sichtweisen auf Softwarevarianten, haben die Verantwortlichen der untersuchten SPL vier voneinander unabhängig modellierte Merkmalmodelle (Anforderungen, Implementierung, Integration und Applikation) gepflegt. Die Größe dieser Merkmalmodelle unterscheidet sich signifikant voneinander. Während ein Merkmalmodell in den Anforderungen eine moderate Anzahl von Merkmalen umfasst, weist ein Merkmalmodell in

der Applikation die fünffache Anzahl an Merkmalen auf.

2.2 Herausforderungen in der Praxis

Abbildung 2 veranschaulicht das Ergebnis der von uns durchgeführten Analyse in vereinfachter und exemplarischer Darstellung am Beispiel einer Steuerungssoftware für ein Autoschiebedach. Das skizzierte Szenario stellt keinen Anspruch auf Vollständigkeit und weist bewusst differenzierte Sichtweisen auf Softwarevarianten auf. Gemeinsame Merkmale sind hinsichtlich einer besseren Übersichtlichkeit ausgeblendet.

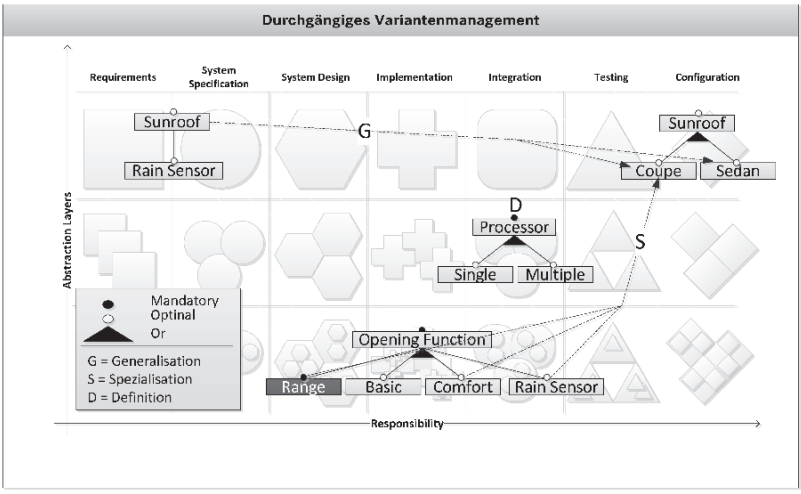


Abbildung 2: Assoziation von Merkmalen im Entwicklungsprozess

Es finden sich Merkmale auf verschiedenen Abstraktionsebenen und in verschiedenen Entwicklungsphasen wieder. Die Definition verschiedener (differenzierter) Merkmale ist auf die externe und interne Produktvariabilität zurückzuführen. Zudem wird die entsprechende Variabilität mit Merkmalen eigenständig dokumentiert und konfiguriert. Die Relevanz eines Merkmals kann daher zwischen den Entwicklungsphasen und Abstraktionsebenen differieren. Eine weitere Erkenntnis unserer Analyse betrifft die eingeschränkte Definition der verwendeten Merkmale. Es werden nur die Merkmale definiert, die in der entsprechenden Verantwortlichkeit ihren Bindezeitpunkt haben. D.h. ein Merkmal dient folglich nur zur Selektion einer bestimmten Variante. Variationspunkte, welche in einer späteren Entwicklungsphase oder in einer anderen Abstraktionsebene gebunden werden, sind nicht durch entsprechende Merkmale dokumentiert. Eine detaillierte Beschreibung der Begriffe Bindezeitpunkt und Variationsmechanismus im Embedded Software Engineering finden sich in [Ro11b].

Lässt man die unterschiedliche Benennung ein und desselben Merkmals außer Acht, sind entsprechende Merkmale nicht vorhanden, auf die sich in einer anderen Abstraktionsebene oder Entwicklungsphase bezogen werden kann. Es zeigt sich beispielsweise, dass ein Merkmal in der Entwicklung weiter verfeinert, zusammengefasst, eingefügt oder gelöscht wird. Ein nachträgliches Harmonisieren isolierter Merkmalmodelle wird somit deutlich erschwert.

Aus unserer Analyse lässt sich allerdings eine Methode zur Realisierung eines durchgängigen Variantenmanagements mittels bestehender Merkmalemodelle ableiten. Operatoren zur Harmonisierung und Assoziation von Merkmalen werden in Kapitel 3 beschrieben.

3 Methodik für ein durchgängiges Variantenmanagement

Gängige Techniken der Variantenmodellierung (z.B. [Ro11a, JBS01, SJ04, PM08]) ermöglichen Assoziationen zwischen Merkmalen und Artefakten herzustellen. Um ein durchgängiges Variantenmanagement über den kompletten Produktlebenszyklus zu ermöglichen, spielt zudem der Umgang mit Merkmalassoziationen eine zentrale Rolle. Im Idealfall ziehen sich die in den Anforderungen definierten Merkmale durch den gesamten Entwicklungsprozess und werden mit den entsprechenden Artefakten verknüpft. Unsere Analyse zeigt, dass im Entwicklungsprozess, differenzierte Merkmale in der Anzahl als auch Semantik vorhanden sind. Folgende Operatoren sind zur Harmonisierung differenzierter Merkmalmodelle geeignet und finden sich sowohl innerhalb als auch zwischen den Abstraktionsebenen und Entwicklungsphasen wieder. In diesem Beitrag fokussieren wir auf Merkmalassoziationen und der damit verbundenen Prozessadaption. Einschränkungen von Merkmalen (z.B. Optional, Mandatory, Requires, OR, AND, etc.) müssen in einem separaten Schritt untersucht werden. Die Operatoren werden im Folgenden definiert und mittels Beispielen illustriert:

1. **O1 (Generalisierung):** Generalisierung erhöht die Variabilität durch Einfügen neuer Merkmale [Ro11a]. Sie entsteht meist durch die technische Realisierung, durch unzureichende Spezifikationen oder durch das nachträglichen Hinzufügen eines Merkmals. Beispielsweise wird das Merkmal *Schiebedach* (Anforderungen) im weiteren Entwicklungsverlauf (Applikation) entsprechend der Dachform eines Autos generalisiert (s. Abb. 2, *G*). So wird ein bestehendes Merkmal mittels mehrerer Merkmale detaillierter beschrieben.

2. **O2 (Spezialisierung):** Spezialisierung reduziert die Variabilität [Ro11a]. Sie entsteht meist durch Einfügen eines übergeordneten Merkmals. Es finden sich beispielsweise die *Komfort-* und *Regensensorfunktion* in einem *Coupé* wieder (s. Abb. 2, *S*). So werden mehrere Merkmale mittels einem Merkmal gruppiert.

3. **O3 (Definition):** Mittels Definition entsteht ein lokales Merkmal ohne Assoziationen zu anderen Merkmalen. Dadurch erhöht sich Variabilität aus einer spezifischen Sicht. Die Definition eines eigenen Merkmals entsteht meist durch voneinander unabhängige Merkmale oder interner Variabilität. Die verwendete Hardware ist bei-

spielsweise nur bei der Integration relevant, da die Implementierung und Applikation hardwareunabhängig erfolgt (vgl. Abb. 2, *D*).

Die beschriebenen Operatoren ermöglichen es Merkmalassoziationen darzustellen. Sie bilden somit die Grundlage für ein harmonisiertes und durchgängiges Variantenmanagement auf Basis bereits bestehender Merkmalmodelle. Unsere Analyse zeigt, dass in der Praxis jeweils nur die Merkmale dokumentiert werden, welche in der entsprechenden Verantwortlichkeit ihren Bindezeitpunkt haben. Merkmale und deren Ausprägungen, die in einer späteren Entwicklungsphase definiert werden, sind mit diesem Vorgehen nur mit großem Aufwand und Expertenwissen mit dem entsprechenden Variationspunkt verknüpfbar. Dadurch entstehen isolierte Merkmale, die ein durchgängiges Variantenmanagement verhindern. Als Beispiel sei die Definition der Merkmale *Coupé* und *Limousine* eines Schiebedaches in der Applikation genannt (s. Abb. 2). Entscheidend für die Applikation ist die Dachform eines Autos, die sich in den Merkmalen *Coupé* und *Limousine* widerspiegelt. Hingegen ist die Dachform in der Implementierung zu vernachlässigen, weswegen eine Verknüpfung der Merkmale ausgeschlossen scheint. Wird die Ursache der Merkmalsaufnahme betrachtet, ist z.B. eine von der Dachform abhängige Öffnungslänge (*Range*) des Schiebedaches zu erkennen, die durch einen Applikationsparameter beeinflusst wird. Der Variationspunkt *Range* wird hierfür bereits in der Implementierung eingefügt, allerdings noch nicht gebunden. Eine direkte Verknüpfung zwischen dem Variationspunkt *Range* und den jeweiligen Ausprägungen *Coupé* oder *Limousine* gestaltet sich in einer umfangreichen SPL hingegen schwierig, da sich die Verantwortlichkeiten unterscheiden oder der Variationsmechanismus und somit der Variationspunkt in der entsprechenden Sichtweise nicht bekannt ist. Aus diesem Grund empfehlen wir die Definition von *indirekten Merkmalen*.

Ein *indirektes Merkmal* wird bei der Realisierung eines Variationspunkts mit einer späteren Bindezeit eingefügt und ist bei der Erstellung *mandatory*. Die Ausprägungen eines *indirekten Merkmals* werden im weiteren Verlauf zur Bindezeit festgelegt. Ein *indirektes Merkmal* beschreibt einen Anknüpfungspunkt für Merkmalassoziationen anderer Entwicklungsphasen oder Abstraktionsebenen. Beispielsweise entsteht durch das Hinzufügen des Merkmals *Range* in der Implementierung (s. Abb. 2, grau hinterlegt) eine sinnvolle Verknüpfung zwischen Dachform und Funktionssoftware. Aus dieser Erkenntnis beschreiben wir folgende Methodik:

1. **M1 (Merkmaldefinition):** Softwarevarianten werden in einem Merkmalmodell auf der höchsten Abstraktionsebene und der ersten Entwicklungsphase dokumentiert. Darauf basierend werden im Entwicklungsprozess neue Merkmale definiert, falls dies keine Redefinition darstellen. Unterschiedliche Ausdrucksweisen und Benennungen für dasselbe reale Merkmal, wie sie in der Praxis auftreten, müssen auf ein einziges Merkmal harmonisiert werden.

2. **M2 (Merkmaldefinition bei Variationspunkten mit späterer Bindzeit):** Es müssen *indirekte Merkmale* für Variationspunkte eingefügt werden, die in einer späteren Entwicklungsphase, einer anderen Abstraktion oder einer verschiedenen Verantwortlichkeit gebunden werden. Entsprechende Merkmalausprägungen des Variationspunkts können später beim Binden der Variabilität hinzugefügt werden.

3. **M3 (Durchgängige Merkmal Assoziation):** Werden neue Merkmale definiert, müssen unter Verwendung der Operatoren O1, O2 und O3 Assoziationen zwischen Merkmalen abstraktions- und entwicklungsphasenübergreifend eingefügt werden.

Unsere Analyse unterstreicht, die Relevanz einer definierten und integrierten Merkmalmodellierung über alle Entwicklungsphasen und Abstraktionsebenen hinweg. Aus diesem Grund empfehlen wir eine Adaption des bestehenden Entwicklungsprozesses zur zentralen Verankerung der Merkmalmodellierung. Ebenso sollte der bestehende Entwicklungsprozess um eine Merkmals- und Artefaktassoziation erweitert werden. Resultierend aus der Adaption des Entwicklungsprozesses von M1, M2 und M3 werden Weiterentwicklungen in einem durchgängigen Merkmalmodell dokumentiert. Unter Zuhilfenahme der beschriebenen Operatoren und der Methodik wird die fortlaufende Integration bereits realisierter Anforderungen ermöglicht. Eine schrittweise Erweiterung zu einem durchgängigen Variantenmanagement kann somit erfolgen. Ein harmonisiertes Merkmalmodell kann als zentraler Einstieg für Erweiterungen oder Fehlerbehebungen verwendet werden. Weitere Anforderungen an Entwicklungsprozesse im Hinblick auf die Realisierung eines durchgängigen Variantenmanagements sind noch zu spezifizieren.

4 Verwandte Arbeiten

Variantenmanagement in SPL wird in der Literatur umfassend beschrieben [PBL05, WL99, TH02, CHE05].

Wird ein durchgängiges Variantenmanagement über den gesamten Entwicklungsprozess betrachtet, entstehen hingegen weitergehende Herausforderungen. [Bo01] beschreibt in seiner Arbeit offene Punkte von SPL, lässt jedoch Lösungsansätze aus. So bestätigt sich in der untersuchten SPL die fehlende Repräsentation von Merkmalen zwischen den Anforderungen und der Realisierung sowie implizite Abhängigkeiten der Software. Ein harmonisiertes Merkmalmodell bildet die Grundlage für die genannten Herausforderungen und ermöglicht z.B. Change-Impact-Analysen und die Dokumentation impliziter Abhängigkeiten. Des weiteren werden unzureichender Toolsupport, fehlende Methoden zur Selektion von Bindezeitpunkten und Auswahl von Variationsmechanismen beschrieben.

[Ro11a] beschreibt verschiedene Dimensionen von Variabilität sowie die Herausforderung diese in Merkmalmodellen abzubilden. Verschiedene Dimensionen in einem Modell abzubilden führt allerdings zu komplexen und großen Merkmalmodellen. Die Differenzierung der Dimensionen in kleinere Merkmalmodelle verringert die Komplexität, verhindert allerdings Analysen zwischen den Dimensionen. Die beschriebene Modellierungssprache *VELEVET* löst diesen Konflikt und könnte zur Realisierung des in diesem Beitrag vorgestellten harmonisierten Merkmalmodells verwendet werden. Die definierten Operatoren Vererbung, Überlagerung und Aggregation setzen ein Verständnis der Merkmalassoziationen voraus. Auf die Definition von Assoziationen zwischen Merkmalen wird hingegen nicht eingegangen.

[CHE05] beschreibt eine Notation zur Merkmalmodellierung. *Staged configuration* beachtet einen realistischen Entwicklungsprozess mit verschiedenen Entwicklern, Gruppen

und Zeitpunkten. Mit *Multi-level configurations* wird das Teilen verschiedener Abstraktionsebenen in eigenständige Merkmalmodelle vermieden. Als Schwachstelle der Merkmalmodellierung beschreibt er die Definition von Merkmalen auf verschiedenen Abstraktionsebenen und begründet dies in einer fehlenden Richtlinie. In *Multi-level configurations* verwaltet jede Rolle (z.B. Sicherheit, Netzwerktechnik) ihr eigenes Merkmalmodell, das nach definierten Richtlinien zu strukturieren ist. Im Gegensatz zu unserem Beitrag werden Assoziationen zwischen Merkmalen nicht behandelt.

[Re08] beschreibt die Struktur von *Product Sublines (Subline)* und erkennt die Notwendigkeit, verschiedene Sublines in Relation miteinander zu setzen. Für eine hierarchische Strukturierung der Sublines werden *Multi-Level Feature Models* und Relationen zwischen diesen definiert. Eine Methodik zur Assoziation von Merkmalen sowie notwendige Prozessadaptionen werden nicht beschrieben.

[HBR10] beschreibt den Provop Ansatz zum Umgang mit Varianten in Geschäftsprozessen. Die Selektion der jeweiligen Variante erfolgt dabei über Kontexte. Varianten in Geschäftsprozessen unterscheiden sich von den in dieser Arbeit betrachteten Produktvarianten in ihrer Sichtweise. Abhängigkeiten zwischen Varianten in Geschäftsprozessen und Produkten müssen in einer separaten Arbeit behandelt werden.

5 Zusammenfassung und Ausblick

Für zielgenaue Change-Impact-Analysen, einem verbesserten Tracing und einer vereinfachten Fehlerbeseitigung innerhalb einer SPL wird ein durchgängiges Variantenmanagement über den gesamten Entwicklungsprozess und den vorhandenen Abstraktionsebenen benötigt. Um die Anforderungen an ein durchgängiges Variantenmanagement zu verstehen haben wir eine existierende SPL analysiert. Dabei fanden wir voneinander isoliert erstellte Merkmalmodelle vor. Als Hauptgründe zeichnen sich die differenzierten Sichtweisen auf Softwarevarianten im Entwicklungszyklus sowie die fehlende Prozessverankerung der Merkmalmodellierung aus. Zur Harmonisierung der bestehenden Merkmalmodelle werden Operatoren beschrieben, um Assoziationen zwischen den isoliert erstellten Merkmalmodellen herzustellen und somit zu einem einheitlichen Verständnis der Variabilität der SPL zu gelangen. In der analyisierten SPL zeigt sich, dass ein Merkmalmodell nicht vollständig von den Anforderungen abgeleitet werden kann. Technische Einschränkungen während der Realisierung sowie die Applikation (Post-Build Parametrierung) der Software fügen neue Merkmale hinzu. Eine Adaption des bestehenden Entwicklungsprozesses um die Integration des Variantenmanagements über Merkmale ist aus diesem Grund unumgänglich. Zusätzliche Prozesschritte sind notwendig, Merkmale strukturiert einzufügen und diese mit vorhandenen Entwicklungsphasen und Abstraktionsebenen zu assoziieren. Die Verwendung eines harmonisierten Merkmalmodells hinsichtlich eines verbesserten Tracings von Änderungen, Change-Impact-Analysen oder der Fehlerbeseitigung in Bezug auf Varianten muss im Weiteren analysiert werden.

Literatur

[BBM05] Kathrin Berg, Judith Bishop and Dirk Muthig. Tracing Software Product Line Variability: From Problem to Solution Space. In *Proc Annual Research Conf of the South African institute of Computer Scientists and Information Technologists on IT research in developing countries*, Seiten 182–191, 2005.

[Bo01] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk Obbink and Klaus Pohl. Variability Issues in Software Product Lines. In European Software Institute, Hrsg., *Proc 4th Int Workshop on Product Family Eng (PFE-4)*, Seiten 11–19, Bilbao, 2001.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming: Methods, techniques, and applications*. Addison-Wesley, Harlow, 2000.

[CHE05] Krzysztof Czarnecki, Simon Helsen and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[HBR10] Alena Hallerbach, Thomas Bauer and Manfred Reichert. Capturing Variability in Business Process Models: The Provop Approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(6-7):519–546, 2010.

[Ka90] K. Kang, S. Cohen, J. Hess, W. Novak and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Bericht CMU/SEI-90-TR-21, Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213, 1990.

[PBL05] Klaus Pohl, Günter Böckle and Frank Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Berlin und Heidelberg, 2005.

[PM08] Klaus Pohl and Andreas Metzger. Variabilitätsmanagement in Software-Produktlinien. In *Software Engineering : Software Engineering 2008. Fachtagung des GI-Fachbereichs Softwaretechnik 18.-22.2.2008 in München*, Jgg. 121 of *LNI*, Seiten 28–41. GI, 2008.

[Re08] Mark-Oliver Reiser. *Managing Complex Variability in Automotive Software Product Lines with Subscoping and Configuration Links*. Dissertation, TU Berlin, 2008.

[Ro11a] Marko Rosenmüller. *Towards flexible feature composition: Static and dynamic binding in software product lines*. Dissertation, Otto-von-Guericke-Universität, Magdeburg, 2011.

[Ro11b] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm and Gunter Saake. Multi-dimensional variability modeling. In *Proc VaMoS 11: 5th Int Workshop on Variability Modelling of Software-intensive Systems; Namur*, Seiten 11–20. ACM, New York, 2011.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management: Software Variability Management. *Science of Computer Programming*, 53(3):259–284, 2004.

[TH02] Steffen Thiel and Andreas Hein. Modeling and Using Product Line Variability in Automotive Systems. *IEEE Software*, 19:66–72, 2002.

[JBS01] van Jilles Gurp, Jan Bosch and Mikael Svahnberg. *On the Notion of Variability in Software Product Lines*. Department of Soft Eng and Computer Science/Blekinge Institute of Technology, 2001.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: A family-based software development process*. Addison-Wesley, Reading and Mass, 1999.

Konzepte zur Erweiterung des SPES Meta-Modells um Aspekte der Variabilitäts- und Deltamodellierung

Peter Manhart[1], Pedram Mir Seyed Nazari[2], Bernhard Rumpe[2], Ina Schaefer[3], und Christoph Schulze[2]

[1]Software-Variantenmanagement, Daimler AG, http://www.daimler.com
[2]Software Engineering, RWTH Aachen, http://www.se-rwth.de
[3]Software Engineering and Automotive Informatics, TU Braunschweig, http://www.tu-bs.de/isf

Abstract: In diesem Beitrag werden Konzepte zur Erweiterung eines mehrperspektivischen Meta-Modells (am Beispiel des SPES Meta-Modells) um Aspekte der Variabilitätsmodellierung durch das Konzept der Delta-Modellierung vorgestellt. Die Konzepte werden exemplarisch anhand der logischen und der funktionalen Perspektive des SPES Meta-Modells illustriert. Die vorgestellten Konzepte können ohne Weiteres auch auf die Anforderungs- und die technische Perspektive angewendet werden. Eine Besonderheit dabei sind die Cross-Cutting-Deltas. Diese gruppieren Deltas der einzelnen Perspektiven und ermöglichen somit Deltas über mehrere Perspektiven hinweg.

1 Einleitung

Das SPES Meta-Modell [Po12] basiert auf einer schrittweisen Verfeinerung der Systembeschreibung und wurde im Rahmen des vorangegangenen Forschungsprojekts SPES 2020 entwickelt. Ziel dieses Forschungsprojektes ist die durchgängig modulare und modellbasierte Entwicklung von eingebetteten Systemen. Dabei wird das System aus unterschiedlichen Perspektiven (oft auch Viewpoints genannt) - Anforderungs-, funktionale oder logische Perspektive - und auf unterschiedlichen Abstraktionsebenen betrachtet. Diese Aufteilung (Abstraktionslevel und Perspektiven) teilen das SPES Meta-Modell in eine (SPES-)Matrix ein (siehe Abbildung 1). Das Meta-Modell fordert nicht ein, dass auf jedem Granularitätslevel und in jeder Perspektive Systemelemente spezifiziert werden. Das bedeutet, dass ein Modell in Level n beispielsweise auch erst 3 Level tiefer verfeinert werden kann. Auch muss kein Modell einer bestimmten Perspektive in den anderen Perspektiven auf demselben Level existieren. Das SPES Meta-Modell erlaubt es, eine Vielzahl von verschiedenen Sprachen, wie zum Beispiel Sequenzdiagramme, Tabellen, Aktivitätsdiagramme und Zustandsdiagramme, zu verwenden. Des Weiteren sind die Zellen der SPES-Matrix in der Regel modular, d.h., diese können unabhängig voneinander entwickelt werden. Außerdem sind weitere Erweiterungen des Meta-Modells, sowohl um Perspektiven (Viewpoints) als auch um neue Sprachen innerhalb der Perspektiven, im Zuge weiterer Forschungsarbeiten beabsichtigt.

In diesem Artikel wird ein Ansatz vorgestellt, um das SPES Meta-Modell - beispielhaft für mehrperspektivische Meta-Modelle mit beliebig vielen Abstraktionsleveln und Sprachen - um Aspekte der Variabilitätsmodellierung durch das Konzept der Delta-Modellierung zu erweitern. Die Delta-Modellierung [CHS10, Sc10] ist dafür gut geeignet, da sie eine modulare und sprachunabhängige Möglichkeit darstellt, die Variabilität von Softwareartefakten zu beschreiben. Eine Systemfamilie wird dabei durch ein ausgezeichnetes Kernsystem und eine Menge von Deltas beschrieben. Das Kernsystem ist eine vollständige Systemvariante, die mit bewährten Prozessen entwickelt werden kann, um ihre Qualität sicherzustellen. Ein Delta beschreibt Modifikationen des Kernsystems, um weitere Systemvarianten zu realisieren. Modifikationen sind in der Regel das Hinzufügen, Entfernen, Verändern oder Ersetzen von Systemelementen. Durch Anwendung einer Menge von Deltas auf das Kernsystem können diese Systemvarianten automatisch erzeugt werden. Eine konkrete Variante ist dann jeweils durch das Basismodel und einer konkreten Kombination von Deltas gegeben. Dies erlaubt auch die Wiederverwendung einzelner Deltas, um unterschiedliche Varianten zu beschreiben. Zum Beispiel kann die grundsätzliche Funktionalität einer Steuerung für die Innenraumbeleuchtung eines Autos als Basismodell definiert werden. Diese schaltet abhängig vom Zustand des zugehörigen Lichtschalters das Licht an oder aus. Eine Steuerung, die unabhängig vom Zustand des Lichtschalters auch bei Öffnung der Tür das Licht einschaltet, wird dann als Variante des Basismodells definiert. Dazu ist es nur nötig, die zusätzliche Funktionalität und die Modifikationen der Basisfunktionalität in einem Delta zu beschreiben.

Der weitere Aufbau dieses Papiers ist wie folgt: Einen Überblick über verwandte Arbeiten gibt Abschnitt 2. Abschnitt 3 beschreibt kurz die Idee zur Erweiterung des SPES-Meta-Modells um Konzepte der Delta-Modellerierung und wird in Abschnitt 4 exemplarisch für Deltas innerhalb einer Perspektiven als auch über mehrere Perspektiven hinweg angewendet. In Abschnitt 5 wird das Konzept anhand eines konkreten Beispiels veranschaulicht. Abschließend fast Abschnitt 6 den vorgestellten Ansatz zusammen.

2 Verwandte Arbeiten

Variabilitätsmodelle können den Problemraum eines Systems, d.h., die Interessen und Wünsche der Stakeholder, oder den Lösungsraum erfassen. Lösungsraumvariabilität (*solution space variability* [Me07]) behandelt die Variabilität von wiederverwendbaren Artefakten, wie Architekturelementen oder anderen Entwicklungsdokumenten. Im Folgenden werden Variabilitätsmodellierungsansätze, für den Lösungsraum betrachtet, da sich der in diesem Artikel vorgestellte Ansatz auf den Lösungsraum bezieht. Die verschiedenen existierenden Ansätze eignen sich unterschiedlich gut zur Erweiterung der SPES Meta-Modells um Aspekte der Variabilitätsmodellierung. Die zwei wichtigen Einflussfaktoren dabei sind die Sprachunabhängigkeit und Modularität eines Ansatzes.

Annotative Ansätze (*annotative approaches*) fassen alle Varianten in einem einzigen Modell (oft 150%-Modell genannt) zusammen, was eine feingranulare Repräsentation der Unterschiede zwischen den einzelnen Varianten erlaubt [Gr08]. Variantenannotationen, wie zum Beispiel UML-Stereotypen [Go05] oder Presence Conditions [Cz05], definieren

welche Teile des Modells entfernt werden müssen, um ein konkretes Modell eines Produkts zu erhalten. Beim Orthogonal Variability Model (OVM) [PBL05] wird Variabilität in einem Variabilitätsmodell, das von den Artefakten getrennt ist, modelliert. Zwischen dem OVM und Artefakten werden explizit Verbindungen gezogen. Ist eine Variante nicht ausgewählt, werden die dazugehörigen Artefakte entfernt. Annotative Ansätze sind grundsätzlich sprachunabhängig, jedoch wird Variabilität monolithisch und nicht modular beschrieben. Damit sind sie eingeschränkt für das SPES-Meta-Modell verwendbar.

Kompositionale Ansätze (*compositional approaches*) zerlegen die Variabilität eines Systems in geeignete Systemfragmente. Sie verknüpfen diese Fragmente mit Produktfeatures, so dass die Fragmente zu einer bestimmten Featurekonfiguration durch den gewählten Kompositionsmechanismus zusammengesetzt werden können. Dabei werden z.B. aspektorientierte Mechanismen [HW07, NK08, VG07] zur Modellierung von Variabilität verwendet. Im feature-oriented model-driven development (*FOMDD*) [TBD07] werden Modellfragemente, die mit einem Produktfeature verknüpft sind, zu Featuremodulen zusammengefasst. In diesen Featuremodulen können Modellierungselemente hinzugefügt oder verfeinert werden. Für eine bestimmte Featurekonfiguration werden die entsprechenden Featuremodule nach den Prinzipien der schrittweisen Verfeinerung [BSR04] kombiniert. Kompositionale Ansätze beschränken die Modellierung der Variabilität auf den gewählten Kompositionsmechanismus. Sie sind daher nicht sprachunabhängig und ebenfalls nur eingeschränkt zur Erweiterung des SPES-Meta-Modells geeignet.

Bei transformationalen Ansätzen (*transformational approaches*) wird die Variabilität eines Systems durch die Transformation eines Basismodells repräsentiert, um eine Produktvariante zu erhalten, wie z.B. in der Common Variability Language (*CVL*) [Ha08]. Der in diesem Beitrag verwendete Ansatz der Delta-Modellierung ist ebenfalls ein transformationaler Ansatz. Ein Beispiel für die Delta-Modellierung von Softwarearchitekturen ist die Architekturbeschreibungssprache Δ-MontiArc [Ha11a, Ha11b]. Transformationale Ansätze, wie die Delta-Modellierung, sind modular und sprachunabhängig, so dass diese sich ausgezeichnet für die Erweiterung des SPES-Meta-Modells eignen.

3 Delta-Modellierung im SPES Meta-Modell

Die Delta-Modellierung zur Variantenbeschreibung lässt sich aufgrund ihrer Sprachunabhängigkeit und ihrer Modularität konzeptuell sehr gut in das SPES Meta-Modell integrieren. Dabei haben wir das Meta-Modell so erweitert, dass Deltas genutzt werden können, um Variabilität innerhalb einer Perspektive und über Perspektiven hinweg zu beschreiben (siehe Abbildung 1). Da innerhalb der SPES-Matrix zwischen einzelnen Artefakten unterschiedlicher Abstraktionsebenen und auch zwischen Artefakten unterschiedlicher Perspektiven Abhängigkeiten bestehen können, ist es notwendig, diese nach Anwendung von Modifikationen durch Deltas konsistent zu halten. Dies bedeutet, dass für jede Modifikation, die Elemente verändert, entfernt oder einführt, die in Abhängigkeit zu Elementen anderer Artefakte stehen, entsprechende Modifikationen auch an den jeweils anderen Artefakten durchgeführt werden müssen, um das gesamte System konsistent zu halten.

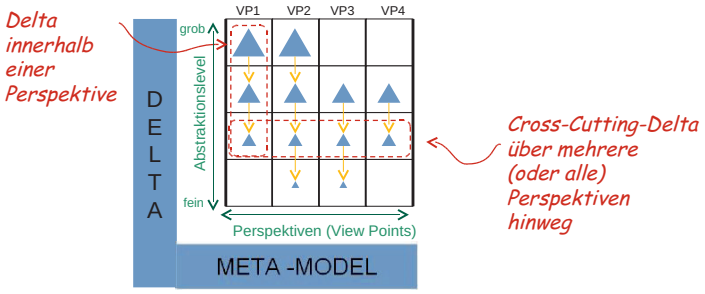


Abbildung 1: Delta-Meta-Modell als Querschnitt im SPES-Meta-Modell

Jede Perspektive der SPES-Matrix kann durch Konzepte der Delta-Modellierung erweitert werden, um Deltas für die jeweiligen Artefakte der Perspektive, aber auch Deltas, welche sich auf unterschiedliche Artefakte einer oder mehrerer Perspektiven gleichzeitig auswirken, zu definieren. Dies erlaubt auch Referenzen zwischen Artefakten und zwischen Perspektiven zu modifizieren. Dabei wird die Variabilität für alle Perspektiven konzeptionell gleichartig beschrieben. Die Variabilität unterschiedlicher Perspektiven kann in einem gemeinsamen Delta (siehe Abschnitt 4) zusammengefasst werden, um gleichzeitig Systemvarianten für unterschiedliche Perspektiven zu erzeugen.

Abbildung 2 zeigt das Delta-Meta-Modell (weiße Klassen, links), welches die syntaktische Struktur von Deltas vorgibt. Jedes Delta wird in einer der beiden Unterklassen der abstrakten Klasse `DeltaModel` definiert.

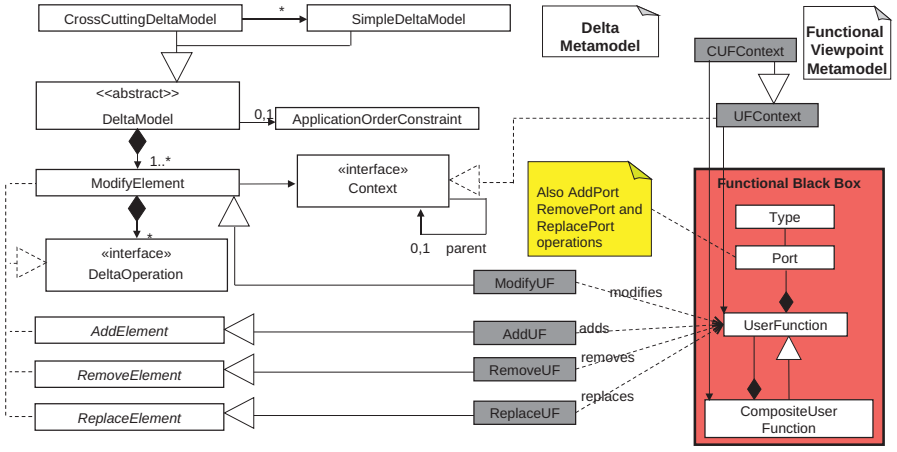


Abbildung 2: Delta-Meta-Modell (l.) und beispielhafte Erweiterung des funktionalen VPs

Deltas innerhalb einer Perspektive werden in `SimpleDeltaModel` und Deltas über

mehrere Perspektiven hinweg in `CrossCuttingDeltaModel` spezifiziert. Jedes Delta enthält mindestens ein `ModifyElement`, welches den zu modifizierenden Kontext (`Context`) referenziert. Ein Kontext gehört zu Elementen, die innere Elemente enthalten können, und kann ebenfalls hierarchisch dekomponiert sein. Auf diese Weise ist es möglich, innere Elemente einer Hierarchie zu modifizieren (zum Beispiel durch Löschen oder Hinzufügen). Modifikationen des Kontexts sind als `DeltaOperations` spezifiziert und beinhalten das Hinzufügen (`AddElement`), Löschen (`RemoveElement`), Ersetzen (`ReplaceElement`) und Modifizieren (`ModifyElement`) von Elementen. Des Weiteren kann ein Delta einen `ApplicationOrderConstraint` (AOC) enthalten, um explizit Abhängigkeiten zwischen Deltas zu modellieren. Dieser wird in Fällen benötigt, in denen Deltas dasselbe (Kern-)Modell modifizieren oder Produktfeatures realisieren, die voneinander abhängen. Ein `CrossCuttingDeltaModel` enthält zusätzlich `SimpleDeltaModels` (siehe Abschnitt 4).

4 Exemplarische Erweiterung von ausgewählten SPES-Perspektiven

Deltas innerhalb eines Viewpoints Die Erweiterung des Meta-Modells um Konzepte der Delta-Modellierung innerhalb einer Perspektive wird anhand der funktionalen Perspektive beispielhaft am (vereinfachten) Functional-Black-Box Modell [Po12] beschrieben (siehe 2). Diese Konzepte können ebenfalls für andere Modelle innerhalb der funktionalen und auch anderer Perspektiven angewendet werden. Solche Deltas werden durch `SimpleDeltaModel`-Elemente modelliert (siehe Abbildung 2).

Wir schlagen folgende (grobe) Vorgehensweise zur Erweiterung einer Modellsprache *M* um Konzepte der Delta-Modellierung vor:

1. Für jedes konzeptuelle Sprachelement *E* einer Modellsprache *M* ist zu bestimmen, ob es modifizierbar ist bzw. sein soll (weiter mit *(2)*) oder nicht (weiter mit *(4)*). Beispiel: In Abbildung 2 ist das Element `Type` nicht modifizierbar[1]. Ein `Type` kann somit nicht über Delta-Operationen ersetzt, gelöscht oder hinzugefügt werden, zumindest nicht direkt (siehe unten).

2. Soll *E* modifizierbar sein, müssen die Operationen (typischerweise Einfügen, Entfernen oder Ersetzen) festgelegt werden, die auf *E* anwendbar sind. Soll *E* ersetzbar sein, ist eine Unterklasse von `ReplaceElement` zu erstellen, welche die Ersetzen-Operation für *E* darstellt. Analog gilt dies für die Hinzufügen- (`AddElement`) und Löschen- (`RemoveElement`) Operationen. Beispiel: Das Element `Port` aus Abbildung 2 ist beispielsweise ersetzbar, weshalb es ein entsprechendes Element `ReplacePort` gibt. Handelt es sich bei *E* um ein komplexer aufgebautes Element, das heißt, ein Element, welches weitere Elemente enthalten kann, weiter mit *(3)*, ansonsten weiter mit *(4)*.

3. Sollen die inneren Elemente eines komplexeren Sprachelements *E* nicht explizit modifiziert werden (ohne dabei *E* komplett austauschen zu müssen), weiter mit *(4)*. Bei-

[1] Dies ist lediglich eine Designentscheidung und soll als Beispiel zum Verständnis des Ansatzes beitragen.

spiel: Dies kann in Abbildung 2 bei `Port` der Fall sein. Ein `Port` hat zwar einen `Type`, allerdings wurde aus methodischen Überlegungen entschieden, dass dieser nicht ausgetauscht werden können soll. Um den Typ eines Ports auszutauschen, muss somit der ganze Port über `ReplacePort` ausgetauscht werden. Ist eine Modifikation innerer Elemente möglich, wird für *E* ein Kontext *C* erstellt, welcher das Interface `Context` aus Abbildung 2 implementiert. Besitzt eine Unterklasse *E_Sub* von *E* weitere - nicht in *E* enthaltene - modifizierbare Elemente, muss ebenfalls ein Kontext *C_Sub* für *E_Sub* erstellt werden, welcher eine Unterklasse von *C* ist. Zusätzlich ist die Erstellung einer Unterklasse von `ModifyElement` notwendig, welche die Modifikations-Operation für *E* darstellt. Zusätzlich kann, beispielsweise über OCL-Constraints, festgelegt werden, welche Delta-Operationen innerhalb des Kontextes *C* erlaubt sind.

Beispiel: In Abbildung 2 sind solche hierarchische Elemente `UserFunction` und `CompositeUserFunction`. Letzteres ist eine Unterklasse des Ersteren und kann zusätzlich zu Ports andere (`Composite-`)`UserFunctions` enthalten. Diese Vererbungsstruktur wird ebenfalls in den beiden Kontexten `UFContext` und `CUFContext` widergespiegelt. Die Modifikations-Operation `ModifyUF` eignet sich sowohl für `UserFunction` als auch für `CompositeUserFunction`. Eine eigene Modifikations-Operation für `CompositeUserFunction` ist nicht notwendig, da abhängig vom aktuellen Kontext (`UFContext` oder `CUFContext`) mittels Constraints bestimmt werden kann, ob Delta-Operationen zum Hinzufügen (`AddUF`), Löschen (`RemoveUF`) und Ersetzen (`ReplaceUF`) von `UserFunctions` erlaubt sind (wenn `CUFContext`) oder nicht. Entsprechende Delta-Operationen gibt es für die Ports einer `UserFunction`.

4. Gibt es weitere Sprachelemente in *M*, welche noch nicht betrachtet wurden, weiter mit *(1)*, ansonsten ist die Spracherweiterung fertiggestellt.

Zu beachten ist, dass dieses grobe Verfahren unbedingt ergänzt werden sollte um spezifische Überlegungen, die in weiteren Operatoren münden. Zum Beispiel sind Refactoring-artige Operatoren sinnvoll, die an mehreren Stellen gleichzeitig Typen ersetzen und so per Konstruktion Konsistenz sichern. Weitere Operatoren könnten die Expansion oder Kapselung von Teilstrukturen ermöglichen, etc.

Im obigem Ansatz bleibt die Basissprache *M* bei der Erweiterung unverändert, stattdessen werden für die Spracherweiterung im Meta-Modell Adaptoren (zum Beispiel `UFContext`) verwendet, um eine Verknüpfung zwischen dem Delta-Meta-Modell und dem Basis-Meta-Modell herzustellen. Das hat unter anderem den Vorteil, dass für die Basismodelle keine Abhängigkeiten zu den Delta-Modellen entsteht.

Cross-Cutting-Deltas Im vorherigen Abschnitt wurde beschrieben, wie das SPES Meta-Modell für einen Modelltyp innerhalb einer Perspektive um Delta-Konzepte erweitert werden kann. Dieser Abschnitt beschäftigt sich mit der Modellierung von Cross-Cutting-Deltas. Charakteristisch für solche Deltas ist, dass sie sich in der Regel auf zwei oder mehrere Artefakte unterschiedlichen Modelltyps beziehen. Somit eignen sie sich sowohl für Artefakte unterschiedlichen Typs innerhalb derselben Perspektive, aber auch für die

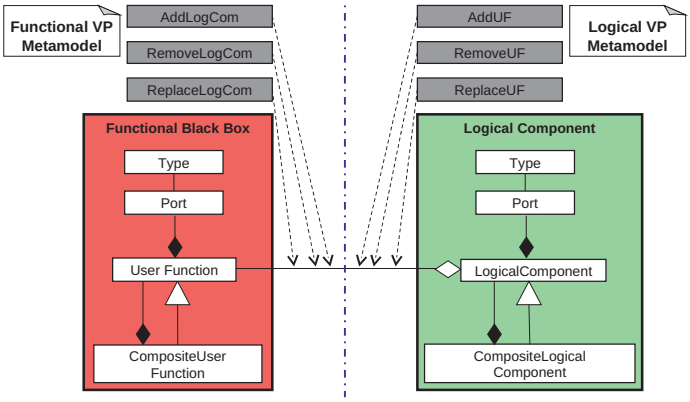


Abbildung 3: Cross-Cutting Delta am Beispiel zweier Perspektiven

Modellierung von Abhängigkeiten in unterschiedlichen Perspektiven. Im Folgenden werden Cross-Cutting-Deltas über mehrere Perspektiven hinweg am Beispiel der funktionalen und logischen Perspektive des SPES Meta-Modells beschrieben.

In Abbildung 3 existiert zwischen `LogicalComponent` aus der logischen Perspektive und `UserFunction` aus der funktionalen Perspektive eine Aggregationsverbindung. Die Bedeutung der gestrichelten Pfeile ist analog zu Abbildung 2. So stellt `AddLogCom` die Verbindung seitens der `UserFunction` her. Die Modifikation der Aggregation betrifft beide Perspektiven. Wird beispielsweise die Aggregation aus `LogicalComponent` entfernt, so muss diese auch aus `UserFunction` entfernt werden, um beide Artefakte konsistent zu halten. Dies wird durch ein `CrossCuttingDeltaModel` (siehe Abbildung 2) mittels entsprechenden Delta-Operationen für die einzelnen Perspektiven (beziehungsweise Artefakte) sichergestellt. Das heißt, ein `CrossCuttingDeltaModel` bündelt (neben den `SimpleDeltaModels`, siehe Abbildung 2) für jede Perspektive die notwendigen Delta-Operationen, um ein Delta über mehrere Perspektiven umzusetzen. In Abbildung 3 wird zum Entfernen der Aggregation die Delta-Operation `RemoveLogCom` auf der funktionalen Perspektive und `RemoveUF` auf der logischen Perspektive benötigt. Die Delta-Operationen `Add-`, `Remove-` und `ReplaceLogCom`, sowie `Add-`, `Remove-` und `ReplaceUF` erben entsprechend der Meta-Modelle in den vorherigen Abschnitten von den Klassen `Add-`, `Remove-` und `ReplaceElement` des Delta-Meta-Modells (aus Platzgründen nicht abgebildet). Ein Cross-Cutting-Delta sorgt dafür, dass diese beiden Deltas auch existieren, da der Zustand inkonsistent wird, wenn nur in einer Perspektive die Aggregation entfernt wird.

Bevor die Cross-Cutting-Deltas bei der Variantengenerierung angewendet werden, werden zunächst alle Deltas, die nur eine Perspektive betreffen (`SimpleDeltaModel`), angewendet, so dass die einzelnen Perspektiven für sich in einem konsistenten Zustand sind. Anschließend folgen die Cross-Cutting-Deltas, um die Konsistenz zwischen den Perspektiven sicherzustellen.

5 Beispiel

Im Folgenden wird das Erstellen einer neuen Variante über mehrere Perspektiven hinweg exemplarisch dargestellt. Das zugrunde liegende Szenario ist die Modellierung zweier Varianten einer Steuerung der Innenraum-Beleuchtung für Autos. Dabei ist die Komplexität der Anforderungen gering gehalten, um die Anwendung von Cross-Cutting-Deltas, bzw. SimpleDeltas, anschaulich darzustellen. Das Kernmodell beschreibt die einfachste Variante der Steuerung, welche es nur ermöglicht, das Licht innerhalb des Autos mittels Schalter ein- und auszuschalten. Die zweite Variante, welche das Licht auch einschaltet, wenn eine Autotür geöffnet ist, wird durch entsprechende (Cross-Cutting-)Deltas modelliert. Sowohl das Kernmodell, als auch die Variante, welche durch Anwendung der Deltas entsteht, sind in Abbildung 4 dargestellt. Dabei sind alle hinzugefügten Elemente blau gekennzeichnet. Elemente, die durch ein Delta entfernt wurden, sind rot gekennzeichnet.

Innerhalb der Anforderungsperspektive wurde jeder Variante ein `HardGoal` zugeordnet, welches durch jeweils eine `UserFunction` realisiert ist. Hierbei entsteht schon die erste Beziehung zwischen zwei Perspektiven: ein `HardGoal` ist in der funktionalen Perspektive als eine `UserFunction` realisiert. Eine weitere Beziehung zwischen zwei Perspektiven entsteht durch die Realisierung einer `UserFunction` durch entsprechende logische Komponenten. Um die Variante `InLightCtrl_DS` einzuführen, ist es wegen dieser Beziehungen zwischen den Perspektiven notwendig, ein Cross-Cutting-Delta zu definieren. Dieses führt in einem ersten Schritt die zugehörigen Modifikationen in jeder Perspektive durch (einfache Deltas, grau unterlegt), um anschließend Deltas anzuwenden, welche die Beziehungen zwischen den Perspektiven modifizieren (Cross-Cutting-Deltas, hellgrau unterlegt). Das Hinzufügen des Goals `InLightCtrl_DS`, welches die angesprochene Anforderung definiert, und dessen Abhängigkeit zum Goal `InLightCtrl`, führt zu einer notwendigen Umstrukturierung innerhalb der funktionalen Perspektive, da die gegebene Funktionalität nun durch eine Komposition zweier Funktionen definiert wird.

Innerhalb der logischen Perspektive wird die definierte Funktionalität durch eine Modifikation der `InLightCtrl` Komponente realisiert. Zur Veranschaulichung ist in Listing 1 der Pseudocode des zugehörigen Deltas angegeben. In Zeile 2 wird die zu modifizierende logische Komponente angegeben und in den folgenden Zeilen werden zuerst notwendige Ports und Komponenten hinzugefügt, um dann abschließend zugehörige Verbindungen

Delta-MontiArc

```
1  delta DoorState {
2      modify component InLightCtrl {
3          add port in Boolean doorStatus;
4          add component Or or;
5
6          connect SwitchStatus -> or.inputA;
7          connect DoorStatus -> or.inputB;
8          connect or.output -> LightCmd;
9      }
10 }
```

Listing 1: Exemplarische Modifikation der Komponente `InLightCtrl` durch ein Delta.

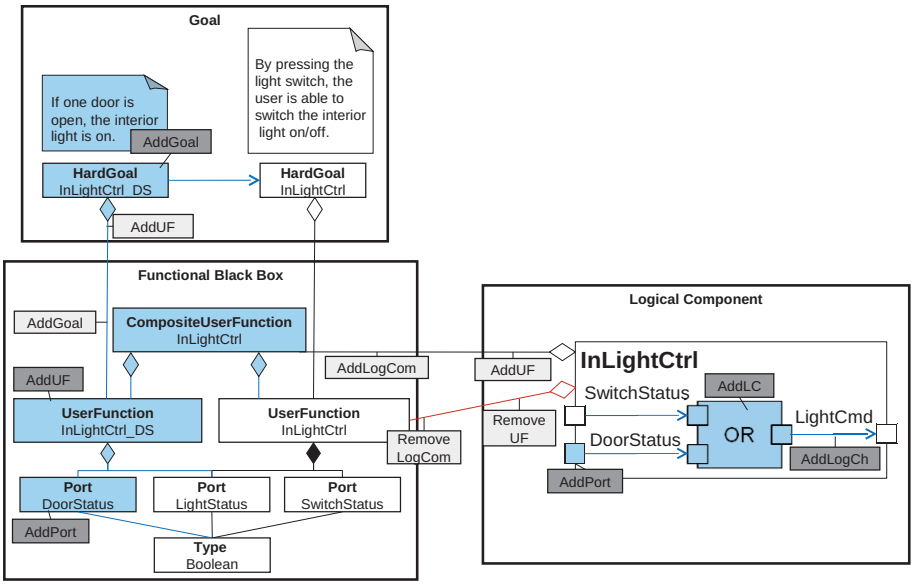


Abbildung 4: Ein Cross-Cutting-Delta über drei Perspektiven.

zu etablieren. Dieses Delta kann grundsätzlich losgelöst von den jeweils anderen Deltas definiert und auch in anderem Kontext verwendet werden.

6 Zusammenfassung

In diesem Artikel wurde ein Ansatz vorgestellt, das mehrperspektivische SPES Meta-Modell um Aspekte der Variabilitätsmodellierung durch das Konzept der Delta-Modellierung zu erweitern. Dabei wurde ein allgemeines Delta-Meta-Modell eingeführt, welches zwischen Deltas innerhalb einer Perspektive (bzw. Modelltyps) und Deltas über mehrere Perspektiven (bzw. Modelltypen) hinweg unterscheidet. Das allgemeine Delta-Meta-Modell kann durch Hinzufügen von Adaptoren für die einzelnen Sprachen des SPES Meta-Modells angepasst werden. Das Konzept wurde anhand eines kurzen Beispiels veranschaulicht. Es ist geplant, dass Konzept an einem größeren Beispiel zu evaluieren, um die Komplexität beim Variantenmanagement mittels Delta-Modellierung zu untersuchen.

Literatur

[BSR04] Don S. Batory, Jacob Neal Sarvela und Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[CHS10] D. Clarke, M. Helvensteijn und I. Schaefer. Abstract Delta Modeling. In *GPCE*. Springer, 2010.

[Cz05] Krzysztof Czarnecki. Mapping features to models: A template approach based on superimposed variants. In *GPCE 2005*, Seiten 422–437. Springer, 2005.

[Go05] Hassan Gomaa. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, 2005.

[Gr08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt und Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, 2008.

[Ha08] O. Haugen, B. Moller-Pedersen, J. Oldevik, G.K. Olsen und A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, Seiten 139 –148, sept. 2008.

[Ha11a] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe und Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *ECSA '11 5th European Conference on Software Architecture: Companion Volume*, New York, NY, USA, September 2011. ACM New York.

[Ha11b] Arne Haber, Holger Rendel, Bernhard Rumpe und Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, Seiten 1 – 10, Munich, Germany, February 2011. fortiss GmbH.

[HW07] F. Heidenreich und C. Wende. Bridging the gap between features and models. *Aspect-oriented Product Line Engineering*, Seite 38, 2007.

[Me07] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens und G. Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, Seiten 243 –253, oct. 2007.

[NK08] N. Noda und T. Kishi. Aspect-Oriented Modeling for Variability Management. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, Seiten 213 –222, sept. 2008.

[PBL05] Klaus Pohl, Günter Böckle und Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Po12] Klaus Pohl, Harald Hönninger, Reinhold Achatz und Manfred Broy. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer, 2012.

[Sc10] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.

[TBD07] S. Trujillo, D. Batory und O. Diaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *ICSE 2007*, Seiten 44 –53, may 2007.

[VG07] Markus Voelter und Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, Seiten 233–242, Washington, DC, USA, 2007. IEEE Computer Society.

Funktionsgetriebene Entwicklung software-intensiver eingebetteter Systeme in der Automobilindustrie – Stand der Wissenschaft und Forschungsfragestellungen[1]

Marian Daun*, Jennifer Brings*, Jens Höfflinger+, Thorsten Weyer*

*Universität Duisburg-Essen
paluno – The Ruhr Institute for Software Technology
Gerlingstr. 16, 45127 Essen
{marian.daun | jennifer.brings | thorsten.weyer}
@paluno.uni-due.de

+Robert Bosch GmbH
Corporate Research, Software
Postfach 30 02 40,
70442 Stuttgart
jens.hoefflinger@de.bosch.com

Abstract: Im Engineering von software-intensiven eingebetteten Systemen in der Automobilindustrie ist ein deutlicher Trend hin zur funktionsgetriebenen Entwicklung erkennbar. Funktionsgetriebene Entwicklung bedeutet, dass die Funktionen des Systems und deren Abhängigkeiten den wesentlichen Bezugspunkt im Engineering darstellen. Die für die Nutzer wahrnehmbaren Funktionen eines Systems (z.B. adaptive Fahrgeschwindigkeitsregelung) werden dabei in Funktionsverbünden realisiert, d.h. durch das zielgerichtete Zusammenwirken verschiedener Systemfunktionen (z.B. Abstandsbestimmung zum vorausfahrenden Fahrzeug, Bestimmung des Soll-Motordrehmoments und Einleiten eines Bremsvorgangs). Auf Basis dieser spezifischen Abstraktion (d.h. Funktionen und deren Abhängigkeiten) werden u.a. Wechselwirkungen zwischen Funktionen analysiert, die Wiederverwendung getrieben oder ein ressourcenoptimiertes Deployment bestimmt. Im Hinblick auf aktuelle Herausforderungen in der funktionsgetriebenen Entwicklung von software-intensiven eingebetteten Systemen in der Automobilindustrie werden im vorliegenden Artikel die Ergebnisse einer Analyse des Stands der Wissenschaft vorgestellt. Auf Grundlage dieser Ergebnisse werden Forschungsfragestellungen skizziert, die in zukünftigen Forschungsaktivitäten adressiert werden sollten.

1 Einleitung

Software-intensive eingebettete Systeme in Fahrzeugen verfügen über eine immer größer werdende Zahl von in Software realisierten Funktionen (vgl. [Br06]). Die einzelnen Software-Funktionen (im Weiteren kurz: *Funktionen*) eines software-intensiven eingebetteten Systems wirken im Betrieb zielgerichtet zusammen, um den Nutzern des Systems (d.h. dem Fahrer eines Fahrzeuges oder den Beifahrern) gegenüber einen Mehrwert zu erbringen. So erbringt etwa ein software-intensives eingebettetes System zur automatischen Erkennung von Verkehrszeichen durch das zielgerichtete Zusammenwirken einzelner Funktionen dieses Systems (z.B. „Auswertung optischer Sensordaten“, „Plausibilitätsprüfung der Verkehrszeichensituation“, „Anzeigen der Verkehrszeichensituati-

[1] Dieser Beitrag wurde im Rahmen des BMBF-Projektes SPES 2020_XT (Förderkennzeichen: 01IS12005C+M) gefördert.

on“) dem Fahrer des Fahrzeuges einen Mehrwert, da dieser zu jedem Zeitpunkt über die auf dem aktuellen Streckenabschnitt geltenden Verkehrsbeschränkungen informiert ist.

Zur Erbringung eines spezifischen Mehrwerts bilden die Funktionen mit den jeweiligen funktionalen Abhängigkeiten einen *Funktionsverbund*. Aus dem Zusammenwirken der Funktionen innerhalb eines Funktionsverbunds resultiert ein an den Systemgrenzen wahrnehmbares Gesamtverhalten. Das Gesamtverhalten eines software-intensiven eingebetteten Systems ist dabei in nicht-trivialen Fällen nur zu Teilen aus einer isolierten Betrachtung des Verhaltens der einzelnen Funktionen des Funktionsverbunds vorhersagbar (vgl. [Pr07]). Bei der *funktionsgetriebenen Entwicklung* von software-intensiven eingebetteten Systemen bilden die Funktionen des zu entwickelnden Systems und deren Abhängigkeiten den wesentlichen Bezugspunkt im Engineering, indem sie u.a. Architekturentscheidungen und das Deployment wesentlich beeinflussen sowie das zentrale Konzept zur systematischen Wiederverwendung sind. Dabei liegt der Schwerpunkt nicht auf den einzelnen Funktionen des Systems, sondern auf dem jeweiligen Funktionsverbund.

Der vorliegende Beitrag zielt darauf ab, Forschungslücken in Bezug auf die funktionsgetriebene Entwicklung von software-intensiven eingebetteten Systemen im Automobilbereich aufzuzeigen und konkrete Forschungsfragstellungen zu formulieren, die zukünftige Forschungsaktivitäten leiten. Hierzu werden in Abschnitt 2 die aktuellen Herausforderungen der Automobilindustrie skizziert, die bezogen auf die funktionsgetriebene Entwicklung bestehen. Abschnitt 3 fasst die Ergebnisse einer durchgeführten Untersuchung des Stands der Wissenschaft in Bezug auf Konzepte, Techniken und Methoden zur Dokumentation und Analyse von Funktionsverbünden zusammen. Auf der Grundlage des Stands der Wissenschaft und der aktuellen Herausforderungen werden in Abschnitt 4 Forschungsfragestellungen formuliert, die in künftigen Forschungsaktivitäten adressiert werden sollten. In Abschnitt 5 werden die Ergebnisse des Beitrags zusammengefasst.

2 Aktuelle Herausforderungen in der Automobilindustrie

Die funktionsgetriebene Entwicklung von software-intensiven eingebetteten Systemen stellt die Automobilindustrie aktuell vor eine Reihe von Herausforderungen, die für die effektive (d.h. Entwicklung eines qualitativ hochwertigen Produkts) und effiziente (d.h. ressourcenschonende) Entwicklung von erheblicher Bedeutung sind (vgl. auch [Pr07]).

2.1 Nachweis emergenter Eigenschaften

Die durch die funktionalen Abhängigkeiten der Funktionen eines Funktionsverbundes induzierten Eigenschaften im Gesamtsystemverhalten des software-intensiven eingebetteten Systems, welche nicht durch eine isolierte Betrachtung des Verhaltens der einzelnen Funktionen vorhersagbar sind, werden als *emergente Eigenschaften* bezeichnet. Emergente Eigenschaften eines software-intensiven eingebetteten Systems können in zwei Kategorien unterteilt werden: (a) *gewünschte emergente Eigenschaften* und (b) *nicht-gewünschte emergente Eigenschaften*. Gewünschte emergente Eigenschaften sind solche emergenten Eigenschaften, die notwendig sind, damit im Zusammenwirken der Funktionen innerhalb eines Funktionsverbunds das System im Betrieb den intendierten

Mehrwert erbringen kann. Demgegenüber sind nicht-gewünschte emergente Eigenschaften solche emergenten Eigenschaften, die zu einem Effekt an den Systemgrenzen führen, der nicht zur Erbringung des Mehrwertes beiträgt, ggf. dessen Erbringung verhindert oder sogar die körperliche Unversehrtheit von Menschen gefährdet.

Bezüglich gewünschter emergenter Eigenschaften besteht die wesentliche Herausforderung darin, den formalen *Nachweis* zu führen, dass der Funktionsverbund die jeweilige emergente Eigenschaft aufweist. Bezüglich nicht-gewünschter emergenter Eigenschaften besteht die zentrale Herausforderung im *Nachweis*, dass der Funktionsverbund keine dieser Eigenschaften zulässt. Beispielsweise muss nachgewiesen werden können, dass im Zusammenspiel der Funktionen einer adaptiven Geschwindigkeitsregelung nicht die Situation eintreten kann, dass die Geschwindigkeit aufgrund eines Vergleichs von Soll- und Ist-Geschwindigkeit von einer Funktion erhöht wird, obwohl von einer anderen Funktion ein vorausfahrendes Fahrzeug erkannt wurde und die Geschwindigkeit von dieser Funktion verringert werden sollte. Eine besondere Schwierigkeit liegt hier in der Berücksichtigung der wechselseitigen funktionalen Abhängigkeiten mit dem Kontext des Systems und den Abhängigkeiten innerhalb des Kontexts. Dies ist gerade in der Automobilindustrie von Bedeutung, da sich emergente Eigenschaften z.B. aus einem Steuer- und Regelkreis ergeben können, der zu großen Teilen im Kontext des Systems liegt.

2.2 Bruchfreie Integration in bestehende Engineering-Ansätze

Für die effektive und effiziente Entwicklung von software-intensiven eingebetteten Systemen existiert mit dem SPES 2020 Modeling Framework [Br12] ein durchgängiger Ansatz, der über verschiedene Granularitätsstufen der Systembetrachtung (d.h. Gesamtsystem, dessen Subsysteme usw.) die systematische Analyse und Spezifikation der Anforderungen, die Definition der notwendigen Systemfunktionen, die Entwicklung der logischen Architektur und den Entwurf der technischen Architektur unterstützt.

Bezüglich der funktionsgetriebenen Entwicklung besteht die wesentliche Herausforderung darin, existierende durchgängige Engineering-Ansätze, wie z.B. das SPES 2020 Modeling Framework, zu erweitern und ggf. anzupassen, um die für die funktionsgetriebene Entwicklung notwendigen Modelltypen (bspw. zur Dokumentation von Funktionsverbünden) bruchfrei in die bestehende Modellierungstheorie integrieren zu können. Dies setzt u.a. voraus, dass die Ontologie, die den neu definierten Modelltypen jeweils zugrunde liegt, auf einem hohen Formalisierungsgrad definiert ist und dass die ontologischen Beziehungen zu den bestehenden Modelltypen formal definiert sind. Diese Form der Durchgängigkeit, d.h. über die verschiedenen Modelltypen, ist notwendig, um z.B. die Konsistenz zwischen dem Funktionsverbund und anderen Entwicklungsartefakten gewährleisten zu können. Ist bspw. in den Anforderungen spezifiziert, dass die adaptive Geschwindigkeitsregelung bei einem Bremseingriff durch den Fahrer sofort zu beenden ist, so muss dies auch vom Funktionsverbund realisiert werden können, z.B. durch das Zusammenspiel einer Funktion zur Überwachung der Pedalstellungen und einer Funktion, die für die Aktivierung und Deaktivierung der adaptiven Geschwindigkeitsregelung verantwortlich ist.

2.3 Partitionierung und Deployment von Funktionsverbünden

Im Zuge des Entwurfs der logischen Systemarchitektur wird der Funktionsverbund eines software-intensiven eingebetteten Systems partitioniert. *Partitionierung* bedeutet, dass die Funktionen des Funktionsverbunds anhand spezifischer Kriterien (wie z.B. Kopplung und Kohäsion der Funktionen) auf Komponenten der logischen Systemarchitektur verteilt werden. Zum Entwurf der technischen Architektur werden dann die Funktionen des Funktionsverbunds zu den zur Verfügung stehenden technischen Ressourcen (genauer: zu Steuergeräten bzw. Prozessoren) allokiert. Generell existieren viele kombinatorische Möglichkeiten zur Allokation der Funktionen zu technischen Ressourcen (d.h. zum *Deployment*), wobei das Spektrum möglicher Deployment-Lösungen durch vorgegebene Qualitätsanforderungen und Rahmenbedingungen (wie z.B. Rechenleistung der verfügbaren Prozessoren, Anforderungen bzgl. Performance und Verlässlichkeit oder die Kapazität der eingesetzten Bussysteme) oftmals stark eingeschränkt wird. So könnte das Wissen, dass es einen hohen Datenaustausch zwischen den Funktionen zur Bestimmung der aktuellen Geschwindigkeit und der Anpassung der Fahrzeuggeschwindigkeit gibt, dazu führen, dass beide Funktionen auf das gleiche Steuergerät allokiert werden, da durch diese Entscheidung das Datenaufkommen auf dem Bus verringert werden kann.

In der funktionsgetriebenen Entwicklung software-intensiver eingebetteter Systeme besteht hinsichtlich der Partitionierung des Funktionsverbundes die wesentliche Herausforderung, eine für den jeweiligen Verwendungszweck der logischen Architektur bestmöglich geeignete Gruppierung der Funktionen des Funktionsverbundes zu logischen Komponenten und deren Abhängigkeiten zu erreichen. Hinsichtlich der Unterstützung des Deployments besteht in der funktionsgetriebenen Entwicklung die wesentliche Herausforderung, unter Berücksichtigung sämtlicher Einschränkungen (d.h. Qualitätsanforderungen, Rahmenbedingungen, Leistungsmerkmale der technischen Ressourcen) eine optimale Lösung für das Deployment zu finden, d.h. etwa *eine* Lösung, die allen Einschränkungen genügt oder in der Menge aller möglichen Lösungen diejenige zu identifizieren, die z.B. die geringsten Herstellungskosten in der Serienfertigung mit sich bringt.

3 Analyse des Stands der Wissenschaft

In diesem Abschnitt werden die wesentlichen Ergebnisse der Untersuchung des Stands der Wissenschaft hinsichtlich der in Abschnitt 2 betrachteten Herausforderungen vorgestellt. Die relevanten Beiträge sind in vier Kategorien von Forschungsarbeiten untergliedert, in jeder Kategorie werden Ansätze vorgestellt, die typische Vertreter der bei der Untersuchung des Stands der Wissenschaft identifizierten Ansätze sind.

3.1 Ansätze zur modellbasierten Dokumentation von Funktionsverbünden

Eine wesentliche Voraussetzung für die Bewältigung der aktuellen Herausforderungen ist die modellbasierte Dokumentation von Funktionsverbünden, bestehend aus hierarchisch strukturierten Funktionen und deren wechselseitigen funktionalen Abhängigkeiten in einem konsistenten Modell. Ansätze zur modellbasierten Dokumentation von Funktionsverbünden betrachten dabei zwei unterschiedliche Funktionsbegriffe: *Nutzer-*

funktionen und *Features* sowie *Systemfunktionen*. Nutzerfunktionen dokumentieren das vom potentiellen Nutzer gewünschte wahrnehmbare Verhalten in abgegrenzten Einheiten (z.B. [BP10] und [Br09]). Nutzerfunktionen beschreiben zumeist das funktionale Verhalten einer Funktion an deren Schnittstellen, wobei Nutzerfunktionen zwecks Komplexitätsreduktion auch dekomponiert werden können. Features werden in der Softwareproduktlinienentwicklung zumeist hierarchisch strukturiert und diese Struktur dokumentiert (vgl. [Ka98]). Darüber hinaus existieren Ansätze, die explizit Abhängigkeiten zwischen Features dokumentieren, um durch Analysen Aussagen über die Auswirkungen auf die Architektur der einzelnen abzuleitenden Systeme treffen zu können (vgl. [Zh06]). Systemfunktionen dokumentieren das vom Entwickler geplante funktionale Verhalten in hierarchischen Strukturen und in Teilen wechselseitige Abhängigkeiten zwischen Funktionen (z.B. [JS00], [Gr08] und [Be07] oder auch [De78]). Wechselseitige Abhängigkeiten werden von den Ansätzen in Form von Nachrichtenaustausch oder Datenflüssen und teilweise auch in Form von Kontrollflüssen dokumentiert. Das Verhalten einer Funktion wird oftmals in einem komplementären Modell dokumentiert (vgl. z.B. [Kl04]).

3.2 Ansätze zur Analyse von emergenten Eigenschaften

Derzeit existieren zwei Arten von Ansätzen, die darauf abzielen, emergente Eigenschaften eines Systems aufzudecken: Ansätze zu *impliziten Szenarien* (z.B. [UKM01], [Le05] und [AEY00]) und Ansätze zum *Feature Interaction Problem* (z.B. [KB98], [FN03] und [SHR07]). Ansätze zu impliziten Szenarien zielen darauf ab, nicht explizit spezifizierte Eigenschaften des Systems, die aber dennoch konsistent zur Systemspezifikation erscheinen, aufzudecken. Aufgedeckte Eigenschaften können in der Folge von Experten bewertet und so in gewünschte und nicht-gewünschte emergente Eigenschaften klassifiziert werden. Die impliziten Szenarien werden in der Regel durch Synthese partieller Interaktionsmodelle zu einem Verhaltensmodell aufgedeckt. Ansätze zum Aufdecken von Feature Interactions beschäftigen sich mit dem ursprünglich aus dem Telekommunikationssektor stammenden Problem als Change-Request-Problem. Ausgangspunkt dabei ist, dass eine existierende korrekte Spezifikation, jeweils um ein einzelne atomare Funktion erweitert wird. Durch wechselseitige Abhängigkeiten dieser Funktion mit anderen Funktionen kann Systemverhalten entstehen, das weder der neuen Funktion noch der existierenden Spezifikation zugeordnet werden kann. Ziel ist es, dieses emergente Verhalten aufzudecken und zu entfernen. Zum Feature Interaction Problem existieren auch spezifische Ausprägungen in der Automobilindustrie (bspw. [Ju08]).

3.3 Ansätze zur bruchfreien Integration in bestehende Engineering-Ansätze

Einschlägige modellbasierte Ansätze zur Integration von Engineering-Modellen betrachten die Auswirkungen von Anforderungsänderungen auf die übrigen Anforderungen oder auf Entwurfsartefakte (vgl. z.B. [YBL11]). Die entsprechenden Ansätze stellen aber zumeist keine Formalismen zur konsistenten Entwicklung der unterschiedlichen Artefakte bereit. Die *sichtenbasierte Entwicklung* betrachtet u.a. Sichtenbildung innerhalb der Anforderungsspezifikation [NKF94] oder innerhalb der Architektur [SF97]. Neben Ansätzen zur Konsistenzprüfung dieser Sichten (vgl. z.B. [FLP99]) existieren auch Ansätze zur konsistenten Entwicklung (z.B. [La08], [SJG07]). Um die bruchfreie Integration zu

ermöglich, existieren Ansätze, die sich mit dem konsistenten Zusammenführen und Zerteilen von unterschiedlichen Sichten beschäftigen (z.B. [Ab08], [SE06], [Xi10] und [Ki10]). Ansätze zum View-Merging sind in der Lage, die verschiedenen Anforderungssichten zusammenzufassen, allerdings betrachtet kein Ansatz die Prüfung gegen funktionale Modelle oder die Transformation in solche. Gleiches gilt für Syntheseansätze, die losgelöst von spezifischen Viewpoints partielle Diagramme und Modelle zu einem einzelnen konsistenten Diagramm zusammenführen (z.B. [WJ10], [UBC09] und [Da09]). Darüber hinaus existieren *Modelltransformationsansätze*, die genutzt werden könnten, um die konsistenten synthetisierten Anforderungen und die Dokumentationsform des Funktionsverbundes in eine Sprache zu übersetzen, in der Konsistenz nachgewiesen werden kann. Es existieren derzeit sowohl modellspezifische Ansätze zur Modelltransformation, bspw. zur Transformation von Zielmodellen in Architekturmodelle (z.B. [Le08] und [Pi12]), als auch generischere Ansätze, die ein Verfahren zur automatischen Transformation eines Ausgangsmodells in ein Zielmodell durch Subtransformationen in *Intermediatemodelle* vorschlagen (vgl. [Mi02]). Zudem existieren Ansätze (z.B. [GF94]), die es erlauben, Nachvollziehbarkeitsinformationen zwischen Elementen verschiedener Sichten zu annotieren sowie Ansätze zur automatisierten Erzeugung dieser Nachvollziehbarkeitsinformationen (z.B. [Su10]).

3.4 Ansätze zur Unterstützung bei Partitionierung und Deployment

Derzeit existiert kein Ansatz, der auf Basis eines dokumentierten Funktionsverbundes und unter Berücksichtigung von wechselseitigen funktionalen Abhängigkeiten die Bestimmung einer optimalen Partitionierung oder eines optimalen Deployments maschinell adressiert. Existierende Ansätze beschäftigen sich z.B. mit der *Analyse von Graphen* [CDS01], untersuchen hierzu u.a. die Abhängigkeiten zwischen den Knoten und versuchen dabei (z.B. mittels gewichteter Kanten), eine optimale Gruppierung dieser Knoten zu bestimmen. Wenn Funktionsverbünde in gerichtete Graphen überführt werden können, ist es denkbar, angepasste graphentheoretische Ansätze als Grundlage zur Bestimmung der optimalen Partitionierung oder eines optimalen Deployments einzusetzen. In diesem Fall sind auch existierende Ansätze zum *Patternmatching* in Graphen relevant. Dabei wird der Graph auf bekannte Muster untersucht und anhand der identifizierten Muster ein Architekturvorschlag entwickelt (z.B. [BNL05] und [GY01]). Da im Bereich eingebetteter Systeme typische Entwurfsmuster, wie bspw. das Model-View-Controller-Pattern, nicht eingesetzt werden können, müssen für das Patternmatching in der Automobilindustrie spezifische Pattern entwickelt werden [KCC04]. Hierzu existieren bereits Ansätze (ebd.), die spezielle Anforderungspattern für eingebettete Systeme entwickeln.

3.5 Zusammenfassende Bewertung des Stands der Wissenschaft

Die Untersuchung des Stands der Wissenschaft hat gezeigt, inwiefern existierende Ansätze die Herausforderungen der funktionsgetriebenen Entwicklung von softwareintensiven eingebetteten Systemen in der Automobilindustrie adressieren. Dabei wurden verschiedene Forschungslücken deutlich. So existiert derzeit kein Dokumentationsformat, das in der Lage ist, Funktionsverbünde, bestehend aus hierarchisch strukturierten Funktionen und wechselseitigen funktionalen Abhängigkeiten zwischen diesen, vollständig in einem konsistenten Modell zu erfassen. Dies ist z.B. notwendig, um eine Prü-

fung der Konsistenz zu anderen dokumentierten Sichten zu ermöglichen oder um die Bestimmung einer optimalen Partitionierung oder ein optimales Deployments des Funktionsverbundes maschinell zu unterstützen. Zudem sind die verschiedenen Ansätze teils formal wenig fundiert (vgl. [BP10]). Funktionale Abhängigkeiten zwischen dem zu entwickelnden System und dessen Kontext werden nicht oder nur unzureichend berücksichtigt. Verfahren zum Nachweis emergenter Eigenschaften existieren derzeit lediglich für die Szenariomodellierung oder zur Aufdeckung von Feature Interactions, welche allerdings die Annahme treffen, dass eine Spezifikation gegen die gleiche Spezifikation geprüft wird, die um genau ein Feature verändert wurde. Der Nachweis emergenter Eigenschaften in Funktionsverbünden wird von allen diesen Verfahren nicht adressiert. Ebenso existieren keine Verfahren, die die Identifikation optimaler Partitionierungen bzw. eines optimalen Deployments von Funktionsverbünden maschinell unterstützen.

4 Forschungsfragestellungen

Durch die Ergebnisse in Abschnitt 3 werden verschiedene Forschungslücken im Hinblick auf die funktionsgetriebene Entwicklung software-intensiver eingebetteter Systeme deutlich, die sich durch spezifische Forschungsfragestellungen konkretisieren lassen.

4.1 Dokumentation von Funktionsverbünden

Es konnten keine Ansätze identifiziert werden, die es gestatten, sämtliche für die Analyse von Funktionsverbünden relevanten Aspekte zu integrieren. Daneben weisen die untersuchten Ansätze Lücken bei der Dokumentation der Wechselwirkungen mit dem funktionalen Kontext und bei der formalen Fundierung auf. Bezüglich der Dokumentation von Funktionsverbünden ergeben sich folgende Forschungsfragestellungen:

- Wie können wechselseitige funktionale Abhängigkeiten zwischen Funktionen in einem integrierten Modell des Funktionsverbundes eines software-intensiven eingebetteten Systems dokumentiert werden?
- Wie können relevante wechselseitige funktionale Abhängigkeiten zwischen den Funktionen im Funktionsverbund eines software-intensiven eingebetteten Systems, zwischen den Funktionen und dem Systemkontexts sowie innerhalb des Systemkontexts dokumentiert werden?

4.2 Nachweis emergenter Eigenschaften

Aktuell existieren keine Verfahren, die es gestatten, gewünschte bzw. nicht-gewünschte emergente Eigenschaften des Funktionsverbunds nachzuweisen. Verfahren zur Identifikation von emergenten Eigenschaften eines Systems existieren allerdings bei der Betrachtung von Szenarien. Neben der fehlenden Anwendbarkeit auf Funktionsverbünde mangelt es an einer ausreichenden Berücksichtigung des funktionalen Kontexts. Bezüglich des Nachweises emergenter Eigenschaften in Funktionsverbünden ergeben sich folgende Forschungsfragestellungen:

- Wie können existierende Verfahren angepasst werden, um den Nachweis gewünschter und nicht-gewünschter emergenter Eigenschaften in Funktionsverbünden von software-intensiven eingebetteten Systemen zu unterstützen?
- Wie können Verfahren zum Nachweis emergenter Eigenschaften in Funktionsverbünden erweitert werden, um wechselseitige funktionale Abhängigkeiten mit dem Kontext und im Kontext eines Systems zu berücksichtigen?

4.3 Bruchfreie Integration in bestehende Engineering-Ansätze am Beispiel des SPES Modeling Frameworks

Existierende Verfahren zum Model-Merging (bzw. zur Synthese von partiellen Modellen) und zur Modell-Transformation können Grundlagen zur bruchfreien Integration in das SPES 2020 Modeling Framework [Br12] liefern. Es wurde jedoch kein Verfahren identifiziert, das unmittelbar die zu den Anforderungen eines software-intensiven eingebetteten Systems konsistente Entwicklung von Modellen des Funktionsverbunds adressiert. Bezüglich der Integration in bestehende Engineering-Ansätze ergeben sich u.a. folgende Forschungsfragestellungen:

- Wie lässt sich die modellbasierte Dokumentation von Funktionsverbünden bruchfrei in die Sichten des SPES 2020 Modeling Framework integrieren?
- Wie können existierende Ansätze zum Model-Merging auf die im SPES 2020 Modeling Framework verwendeten Anforderungsartefakte angewendet werden, sodass ein zur Transformation in das Modell des Funktionsverbunds geeignetes konsistentes Intermediatemodell entsteht?

4.4 Partitionierung und Deployment von Funktionsverbünden

Es wurden keine Verfahren identifiziert, die die maschinelle Analyse von Entscheidungen zur Partitionierung und zum Deployment eines Funktionsverbundes unter Berücksichtigung der wechselseitigen funktionalen Abhängigkeiten unterstützen. Allerdings existieren generelle graphentheoretische Verfahren, die eine Grundlage bilden können. Hinsichtlich der Unterstützung der Partitionierung und des Deployments von Funktionsverbünden ergeben sich folgende Forschungsfragestellungen:

- Welche Auswirkungen haben wechselseitige funktionale Abhängigkeiten im Modell des Funktionsverbundes auf die Güte des Partitionierungsentwurfs hinsichtlich der unterschiedlichen Partitionierungsziele und hinsichtlich der unterschiedlichen Einschränkungen des Deployments?
- Wie können generelle graphenbasierte Analysetechniken spezialisiert werden, um die Auswirkungen von Entscheidungen zur Partitionierung bzw. zum Deployment des Funktionsverbundes messbar und verständlich in einem Analysemodell darzustellen?

5 Zusammenfassung

Der Trend zur funktionsgetriebenen Entwicklung von software-intensiven eingebetteten Systemen stellt das Engineering vor Herausforderungen, die bewältigt werden müssen, damit eine effektive und effiziente Entwicklung solcher Systeme gewährleistet werden kann. Im vorliegenden Artikel wurden wesentliche Herausforderungen der funktionsgetriebenen Entwicklung von software-intensiven eingebetteten Systemen in der Automobilindustrie vorgestellt und der Stand der Wissenschaft hinsichtlich dieser Herausforderungen untersucht. Aufbauend auf diesen Ergebnissen wurden Forschungsfragestellungen formuliert, die künftige Forschungsaktivitäten auf dem Gebiet der funktionsgetriebenen Entwicklung von software-intensiven eingebetteten Systemen bestimmen sollten.

Literaturverzeichnis

[Ab08] Abi-Antoun, M.; Aldrich, J.; Nahas, N.; Schmerl, B.; Garlan, D.: Differencing and merging of architectural views. In: ASE Journal, März 2008. S. 35-74.

[AEY00] Alur, R.; Etessami, K.; Yannakakis: Inference of Message Sequence Charts. In: Proc. of the ICSE, 2000. S. 304-313.

[Be07] Beeck, M.: Development of logical and technical architectures for automotive systems. In: Software Systems Modelling, 2007. S. 205-219.

[BNL05] Beyer, D.; Noack, A.; Lewerentz, C.: Efficient Relational Calculation for Software Analysis. In: TSE, Februar 2005. S. 137-149.

[BP10] Brinkkemper, S.; Pachidi, S.: Functional Architecture Modeling for the Software Product Industry. In: Proc. of the ECSA, 2010, S. 198-213.

[Br06] Broy, M.: Challenges in automotive software engineering. In: ICSE, 2006. S. 33-42.

[Br09] Broy, M.; Gleirscher, M.; Merenda, S.; Wild, D.; Kluge, P.; Krenzer, W.: Toward a Holistic and Standardized Automotive Architecture Description. In: IEEE Computer, 2009. S. 98-101.

[Br12] Broy, M.; Damm, W.; Henkler, S.; Pohl, K.; Vogelsang, A.; Weyer, T.: Introduction to the SPES Modeling Framework. In (Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M. Hrsg.): Model-Based Engineering of Embedded Systems Methodology. Springer, Berlin, Heidelberg, 2012.

[CDS01] Cox, L.; Delugach, H.; Skipper, D.: Dependency Analysis Using Conceptual Graphs. In: Proc. of the ICCS, 2001. S. 117-130.

[Da09] Damas, C.; Lambeau, B.; Roucoux, F.; van Lamsweerde, A.: Analyzing Critical Process Models through Behaviour Model Synthesis. In: Proc. of the ICSE, 2009. S. 441-451.

[De78] DeMarco, T.: Structured Analysis and System Specification. Yourdon, NewYork, 1978.

[FLP99] Fradet, P.; Le Métayer, D.; Périn, M.: Consistency Checking for Multiple View. In: Proc. of the ESEC/FSE, 1999. S. 410-428.

[FN03] Felty, A.; Namjoshi, K.: Feature Specification and Automated Conflict Detection. In: TOSEM, Januar 2003. S. 3-27.

[GF94] Gotel, O.; Finkelstein, C.: An analysis of the requirements traceability problem. In: Proc. of the RE Conf., 1994. S. 94-101.

[Gr08] Grönniger, H.; Hartmann, J.; Krahn, H.; Kriebel, S.; Rothhardt, L.; Rumpe, B.:View-Centric Modeling of Automotive Logical Architectures. In: MBEES, 2008. S. 3-12.

[GY01] Gross, D.; Yu, E.: From Non-Functional Requirements to Design through Patterns. In: RE Journal, 2001. S. 18-36.

[JS00] Jantsch, A.; Sander, I.: On the Roles of Functions and Objects in System Specification. In: Proc. of the 8th Intl. Workshop on HW/SW Codesign, 2000. S. 8-12.

[Ju08] Juarez Dominguez, A.: Feature Interaction Detection in the Automotive Domain. In: Proc. of the ASE, 2008. S. 521-524.
[Ka98] Kang, K.; Kim, S.; Lee, J.; Kim, K.; Kim, G.; Shin, E.: FORM: A feature-oriented reuse method with domainspecific reference architectures. In: Annals of Softw. Eng., Nr. 5, 1998. S. 143-168.
[KB98] Kimbler, K.; Bouma, L.: Feature Interactions in Telecommunication and Software Systems V. In: IOS Press, Amsterdam, 1998.
[KCC04] Konrad, S.; Cheng, B.; Campbell, L.: Object Analysis Patterns for Embedded Systems. In: TSE, Dezember 2004. S. 970-992.
[Ki10] Kimelman, D.; Kimelman, M.; Mandelin, D.; Yellin, D.: Bayesian Approaches to Matching Architectural Diagrams. In: TSE, März/April 2010. S. 248-274.
[Kl04] Klein, T.; Conrad, M.; Fey, I.; Grochtmann, M.: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. In: Proc. of Modellierung, 2004. S. 31-41.
[La08] van Lamsweerde, A.: Requirements Engineering: From Craft to Discipline. In: Proc. of the ACM SIGSOFT/FSE, 2008. S. 238-249.
[Le05] Letier, E.; Kramer, J.; Magee, J.; Uchitel, S.: Monitoring and Control in Scenario-Based Requirements Analysis. In: Proc. of the ICSE, 2005. S. 382-391.
[Le08] Letier, E.; Kramer, J.; Magee, J.; Uchitel, S.: Deriving event-based transition systems from goal-oriented requirements models. In: ASE Journal, 2008. S. 175-206.
[Mi02] Milicev, D.: Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments. In: TSE, April 2002, S. 413-431.
[NKF94] Nuseibeh, B.; Kramer, J.; Finkelstein, A.: A framework for expressing the relationships between multiple views in requirements specification. In: TSE, Okt. 1994. S. 760-773.
[Pi12] Pimentel, J.; Lucena, M.; Castro, J.; Silva, C.; Santos, E.; Alencar, F.: Deriving software architectural models from requirements models for adaptive systems: the STREAM-A approach. In: RE Journal, 2012. S. 259-281.
[Pr07] Pretschner, A.; Broy, M.; Krüger, I.; Stauner, T.: Software Engineering for Automotive Systems: A Roadmap. In: Future of Software Engineering, 2007. S. 55-71.
[Sa07] Sabetzadeh, M.; Nejati, S.; Liaskos, S.; Easterbrook, S.; Chechik, M.: Consistency Checking of Conceptual Models via Model Merging. In: Proc. of RE, 2007. S. 221-230.
[SE06] Sabetzadeh, M.; Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. In: RE Journal, 2006. S. 174-193.
[SF97] Spanoudakis, G.; Finkelstein, A.: Reconciling requirements: a method for managing interference, inconsistency and conflict. Annals of Softw. Eng., 1997. S. 433-457.
[SHR07] Shiri, M.; Hassine, J.; Rilling, J.: Feature Interaction Analysis: A Maintenance Perspective. In: Proc. of the ASE, 2007. S. 437-440.
[SJG07] Seater, R.; Jackson, D.; Gheyi, R.: Requirements progression in problem frames: deriving specifications from requirements. In: RE Journal, 2007. S. 77-102.
[Su10] Sundaram, S.; Hayes, J.; Dekhtyar, A.; Holbrook, E.: Assessing traceability of software engineering artifacts. In: RE Journal, 2010. S. 313-335.
[UBC09] Uchitel, S.; Brunet, G.; Chechik, M.: Synthesis of Partial Behavior Models from Properties and Scenarios. In: TSE, Mai/Juni 2009. S. 384-406.
[UKM01] Uchitel, S.; Kramer, J.; Magee, J.: Detecting Implied Scenarios in Message Sequence Chart Specifications. In: Proc. of the ESEC/FSE, 2001. S. 74-82.
[WJ10] Whittle, J.; Jayaraman, P.: Synthesizing Hierarchical State Machines from Expressive Scenario Descriptions. In: TOSEM, Januar 2010. S. 8:1-8:45.
[YBL11] Yue, T.; Briand, L.; Labiche, Y.: A systematic review of transformation approaches between user requirements and analysis models. In: RE Journal, 2011. S. 75-99.
[Xi10] Xing, Z.: Model Comparison with GenericDiff. In: Proc. of the ASE, 2010. A. 135-138.
[Zh06] Zhang, W.; Mei, H.; Zhao, H.: Feature-driven requirement dependency analysis and high-level software design. In: RE Journal, 2006. S. 205-220.

ME'13 – First European Workshop on Mobile Engineering

Using RenderScript and RCUDA for Compute Intensive tasks on Mobile Devices: a Case Study

Roelof Kemp, Nicholas Palmer, Thilo Kielmann, Henri Bal
VU University
De Boelelaan 1081A
Amsterdam, The Netherlands
{rkemp, palmer, kielmann, bal}@cs.vu.nl

Bastiaan Aarts, Anwar Ghuloum
NVIDIA
2701 San Tomas Expressway
Santa Clara, CA, USA
{baarts, aghuloum}@nvidia.com

Abstract: The processing power of mobile devices is continuously increasing. In this paper we perform a case study in which we assess three different programming models that can be used to leverage this processing power for compute intensive tasks. We use an imaging algorithm and compare a reference implementation of this algorithm based on OpenCV with a multi threaded RenderScript implementation and an implementation based on computation offloading with Remote CUDA. Experiments show that on a modern Tegra 3 quad core device a multi threaded implementation can achieve a 2.2 speed up factor at the same energy cost, whereas computation offloading does neither lead to speed ups nor energy savings.

1 Introduction

Over the last two years we observed that mobile processors not only increased in clock speed, but also made the step to multicore. With the introduction of dual core processors for mobile devices in 2010 and quad core processors in 2011, the available raw compute power increased significantly. When hardware shifts the horizon of compute power, there is always software that takes advantage of it. Today, for example, high end mobile devices offer a gaming experience near or better than console quality.

With the increase of processing power came also an increase in complexity of the mobile hardware, such as the aforementioned multicore processors, but also dynamic frequency and voltage scaling, power gating and more. Whereas some complexity is transparent to software, others require software to be rewritten in order to take full advantage of it, which in turn adds complexity to the software.

This is especially true for multicore processors, that can only unleash their processing power if applications are written to execute in multiple threads. While the default programming languages on the major mobile platforms (Android: Java, iOS: Objective-C,

Windows Phone: C#) offer multi threading by default, it is still up to the developer to use it. Moreover, it is likely that if a certain app really needs performance it will be using either a lower level language – such as C – or a specialized high performance language such as Google's RenderScript [Ren].

Next to performance gains through the use of multi threading and specific languages, it has been noted that over the years the gap in computation power between mobile and non mobile devices got smaller. Despite this improvement, the fundamental constraints of a mobile device with respect to size, heat dissipation and power supply remained and therefore non mobile devices continue to offer more processing power than their mobile counterparts. A technique that takes advantage of this difference is *computation offloading* where heavy computation is transferred to non mobile devices to decrease execution time and/or save energy [LWX01].

In this paper we assess three different programming models; low level language, specialized language and computation offloading, using a case study for a High Dynamic Range photography app on a quad core mobile device. We compare the resulting implementations based on execution time and energy usage.

We find that the specialized compute language implementation leads to speed ups of max 2.2 times on a quad core device at the same energy cost; the use of computation offloading, however, has no benefits both in execution time and energy usage.

The remainder of this paper is organized as follows. Section 2 discusses the hardware and software as well as the programming models that we use in our case study. Then in Section 3 we detail the implementations for multicore and computation offloading, after which we discuss the methodology of our experiments in Section 4. Then in Section 5 we discuss the results of the experiments and we conclude in Section 6.

2 Background

2.1 Hardware

For this study we use the NVIDIA Tegra 3 Developer tablet, the only available mobile quad core processor during our study. The Tegra 3 SoC has a 4-PLUS-1 architecture which has either a low power core active if the load is low, or 1-4 normal cores if the load is higher. The maximum clock speed – 1.4 GHz – of the device can only be reached when one of the four normal cores is active; as soon as multiple cores are active the clock speed is reduced to 1.3 GHz.

On the developer device it is possible to explicitly turn cores on or off with *hotplug*. Furthermore, the device has several power rails on which different components are placed. For each power rail the amperage and power consumption can be read out in real time, both in software and with a special breakout board that can be connected to a regular computer. The NVIDIA Tegra 3 Developer tablet runs the Android 4.0.3 operating system.

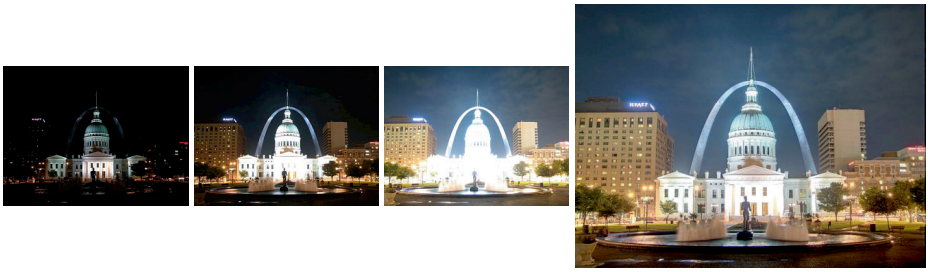


Figure 1: Example of three input images with different exposure levels and the resulting HDR image. Images from http://en.wikipedia.org/High_dynamic_range_imaging

2.2 Software

The application that we focus on during the case study is a demo Camera app – similar to the standard Android 4.0 Camera app – with enhanced photography functionality, such as Negative Shutter Lag and High Dynamic Range (HDR) photography.

More specific, we focus on *exposure fusion*, a computer vision algorithm within the HDR process. This algorithm performs the computationally expensive operation of fusing together multiple images taken from roughly the same spot and the same time with different exposure levels, in such a way that the result image shows details in both dark and bright regions, which can greatly improve the end-user experience of capturing images in scenes with details in both the darker and brighter areas. An example of such a scene and the resulting HDR image is shown in Figure 1.

A detailed specification of the algorithm can be found in [MKVR07]. Relevant for this paper is that the computation in this algorithm is based on matrix, filter and pyramid operations, are all data parallel operations. We use a *control script* around these operations to control how the output of one operation is used as input for another operation.

2.3 Programming Models

Android suppports multiple programming models and languages (see Table 1). Using the Android's Software Development Kit (SDK) Java is the default language, but one can switch to the Native Development Kit (NDK), which uses C/C++ through the Java Native Interface (JNI) for better performance.

Furthermore, with the introduction of multicore processors and the expectation of GPGPU computing becoming available on mobile devices the RenderScript programming model was introduced. RenderScript allows programmers to write kernels that at runtime are automatically run in parallel on the hardware selected by the RenderScript runtime. Until recently RenderScript could only execute code on CPUs, but the Nexus 10 now supports

Table 1: Programming Models on Android

	Language	OpenCV	Automatic Parallelism
SDK	Java	Yes	No
	RenderScript	No	Yes
NDK	C/C++	Yes	No
	OpenGL	No	Yes
Computation Offloading	RCUDA	Yes	Yes

execution on the GPU. Before there was hardware available on which RenderScript can run on the GPU, it was already possible to do general computational jobs on the GPU by writing OpenGL shaders, provided the algorithm and data can be expressed in graphics operations and graphics data.

In addition to the programming models provided by the Android platform that ultimately execute code on the device itself, one can also use a programming model that uses the communication means of a mobile device to offload computation to another better suited device, such as a desktop machine. An example of such a programming model is the Remote CUDA (RCUDA) computation offloading framework.

Of major importance for the subject of our field study, the HDR algorithm, is the availability of the common open source imaging library OpenCV [Ope]. The OpenCV library is written in C++, and has Java wrappers so that it can be used from both the SDK and the NDK on Android. OpenCV is not only used on mobile devices, but also on desktop systems. On desktop systems there are many kernels, for which OpenCV has GPGPU implementations written in CUDA[CUD] to employ data parallelism on the imaging data. With OpenCV's default Android build however, data parallelism is turned off, because no current mobile hardware running Android supports CUDA. However, with the RCUDA computation offloading framework we are able to run the OpenCV library *with CUDA support* on Android in combination with a CUDA enabled device hosting a server. The implementation of RCUDA that we used is similar to the framework described by Duato et al. [DIM+10], however their framework is targeted at HPC cluster systems.

The demo Camera application we use in our field study comes with an implementation built on top of OpenCV in the NDK, which has the drawback of being single threaded. We use this implementation as reference to two new implementations: an implementation with RenderScript that should automatically use all the available cores on the Tegra 3 and an implementation with RCUDA to study the impact of computation offloading. [1]

The focus of the remainder of the paper is on execution times and energy usage of the various implementations. A more qualitative comparison between the SDK, NDK and RenderScript can be found in [QZL12].

[1] Although we have an OpenGL implementation of the algorithm, the mapping of the algorithm to graphics primitives and the limited memory of the GPU resulted in an OpenGL algorithm that does not produce the same results as the others, so we chose to leave this out in the remainder of the paper. We did not make an implementation in Java, because it is essentially the same as the native implementation, however with the addition of overhead due to the Java wrappers in OpenCV if used from the SDK.

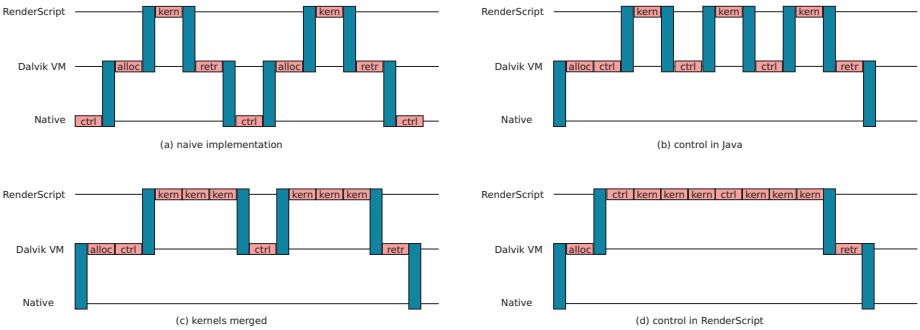


Figure 2: Schematic overview of the optimizations of the RenderScript implementation. ctrl: control script, alloc: memory allocation, kern: execution of a kernel, retr: retrieval of the results. The vertical boxes indicate context switching (JNI) overhead. The boxes are not proportional to the actual execution time.

3 Implementations

3.1 RenderScript

RenderScript is a host/device language similar to languages such as OpenCL and CUDA. RenderScripts host code is written in Java, whereas the device code is written in C99. Memory allocations can only be done in host code. Device code can be started single threaded or multi threaded. If device code is multi threaded it executes a particular kernel for each element of a 1, 2, or 3 dimensional array. RenderScript automatically distributes the elements over the available processors and also does load balancing. While the kernel code is written in C99, it is compiled to an intermediate byte code which, at runtime, gets compiled to the appropriate instructions for the hardware that the RenderScript runtime selects.

To port the reference implementation to RenderScript we initially replaced the OpenCV calls with RenderScript equivalents, while leaving the control code as is. This naive port allowed for easy debugging of the RenderScript kernels because we could switch between OpenCV and RenderScript at kernel level and therefore debug kernels individually instead of debugging the entire algorithm.

Once all kernels were correctly ported we started optimizing the code. The first naive port suffered from the overhead of continuous transitions to different execution environments (see Figure 2-(a)). Each kernel invocation starts in the native environment, then goes to the virtual machine using JNI where Java code is used to retrieve the data from the native environment and allocate it for the RenderScript environment. Once the data is available to RenderScript, the kernel executes (in parallel) and afterwards the Java code retrieves the resulting data from the RenderScript environment and passes it to the native environment. Then the control code selects the next operation and the same process happens over again, until the control code is finished and the final HDR picture is computed.

To reduce the overhead related to the transition from native to Java and back with JNI, we moved the control code to Java such that we only have a single transition to Java at the start of the control function. Then the entire control code is executed in Java and only when the result is computed the program goes back to the native environment (see Figure 2-(b)). By using references to the memory allocations, this implementation did not have to copy intermediate data back and forth to RenderScript, except from the initial input images and the final output image.

We noticed that the transition from Java to RenderScript also added overhead to the algorithm and therefore we started to merge as many kernels together as possible (see Figure 2-(c)). As an example, instead of executing an *add* kernel followed by a *multiply* kernel one can create a single *add-multiply* kernel. This optimization enabled the reduction of the number of kernel invocations drastically, albeit that the kernels themselves are larger.

Finally, we further reduced the transition overhead from Java to RenderScript by moving the control script to RenderScript (see Figure 2-(d)). While the Java code still does all the memory allocations, it will transfer the control to RenderScript which starts computing until the final image is computed without any context switches.

The comparison of the resulting optimized RenderScript implementation versus the reference NDK implementation is discussed in Section 5.

3.2 RCUDA

Next to an implementation with RenderScript that exploits data parallelism on the device itself, we also implemented exposure fusion with computation offloading using Remote CUDA. RCUDA does classic computation offloading by offering a proxy on the client side that forwards calls to a server. RCUDA operates at the CUDA abstraction layer and therefore executes computation on the GPGPU of the host in parallel.

On the mobile device there is a modified version of the CUDA shared library (libcuda.so) that can communicate over TCP with a remote cudaserver. On top of libcuda.so the regular CUDA runtime (libcudart.so) and CUDA libraries can be run (cufft, NPP, etc.).

Porting the exposure fusion algorithm from the reference NDK implementation to a RCUDA implementation is trivial. All the OpenCV kernels that are used in the NDK already have a CUDA based implementation, only the package names differ between the regular and the GPGPU implementation – cv::<kernel> for the regular and cv::gpu::<kernel> for the GPGPU implementation. Furthermore, the data type of the matrices the kernels operate on have to be changed from cv::Mat to cv::gpu::GpuMat and these matrices have to be initialized somewhat differently using a function called upload, because data now resides in GPU memory. Data can be retrieved from a GpuMat using the download function.

The exposure fusion algorithm is the first algorithm of substantial size that has been used with RCUDA and therefore we expected to identify performance bottlenecks in RCUDA. Because the computation offloading in RCUDA is at the CUDA call abstraction level, a reasonably sized algorithm, such as our exposure fusion algorithm, can easily include

thousands of calls. For each call a synchronous request is sent to the server that, depending on the call, immediately returns a response and executes the request asynchronously, or first executes the request and thereafter returns a response. Either way the client blocks until the response arrives.

Because of the sheer number of calls, even a low network latency of 1 ms would add communication overhead of multiple seconds to the process. Therefore we changed as many requests as possible from synchronous to asynchronous, we cached the results of some calls that were called repeatedly on the client side, thereby reducing the number of CUDA calls over the network and the overhead of the calls. Furthermore we noticed that for the remaining synchronous calls, Nagle's algorithm [Nag84] in TCP – buffering small messages into a large message for a certain time – caused much overhead, as described in [DIM+10]. Turning this off with TCP_NODELAY increased performance dramatically.

In addition to the above latency based optimizations to RCUDA we also introduced compression to the client-server protocol, to reduce the amount of data that needs to be transferred at the cost of a additional computation. We use the zlib compression library, which supports 10 different compression levels, ranging from easy compression at a low cost to very complex compression at a high cost.

In the experiments section (Section 5) we assess the impact of latency, bandwidth and compression on the execution time and energy usage of the RCUDA implementation of the exposure fusion algorithm on a Tegra 3 device.

4 Methodology

4.1 Targets and Variables

The key targets of our experiments are the execution time and the energy usage of a particular implementation under particular circumstances. Next to these main targets, we also collect data about the CPU load, the CPU frequency and the temperature, to be able to investigate unexpected results. With this information we can for instance see if using multiple cores indeed leads to all processors being active, or if temperature causes the frequency to be scaled down thereby lengthening execution time.

Next to a specific implementation there are several other variables that will impact the execution time and energy usage of the exposure fusion algorithm. The more pixels an image has, the longer the execution takes; the higher the latency or the lower the bandwidth, the longer computation offloading takes. The more complex the compression, the more computation is required, but also the less data has to be sent.

For varying the latency we artificially increase the latency on the server side using *netem* [H+05]. With *trickle* [Eri05] we manipulated the bandwidth for both the up and downlink of the server. We used the *hotplug* feature of linux to measure the impact of the number of active cores for the multicore implementation.

For each combination of settings we repeated the experiment 30 times. Whereas the results

in general are very consistent, we inspect the data for explainable outliers. We use the additional information such as temperature, CPU usage and CPU frequency to determine whether we discard the outlier for the final results. In all experiments we have at least 26 valid data points.

For the RCUDA experiments we remove the result of the first execution, because it includes a one time overhead of initializing the libraries. Furthermore, for the RCUDA experiments we use ethernet over USB to connect the mobile device to the network, to prevent interference artifacts from the wireless network and make the experiments repeatable.

The Tegra 3 Developer Tablet supports hardware monitoring through various monitoring applications in combination with a PM292 breakout board. The main advantage of hardware monitoring is that it does not interfere with a running application, it does not take cycles from the CPU nor consumes energy from the battery. The main disadvantage of using hardware monitoring is that we cannot easily correlate the data we gather with the execution of a particular part of an application, because the gathered data will be timestamped with the time on the external machine, not the tablet. To overcome this issue we can use software monitoring, such as Power Tutor [ZTQ+10]. In our experiments we use software monitoring where we read out the current power consumption values from the power rail that hosts the quad core.

5 Experiments

5.1 Multi Core

In our first experiment we compare the execution times of the NDK implementation and the RenderScript implementation, while varying the image size and the number of active cores. The RenderScript execution times include both the (serial) host code and (parallel) device code. Since the computation scales with the number of pixels we expect a linear relation between the image size and the execution time. Furthermore, we expect RenderScript to perform up to 4 times better than the NDK implementation, because it can make use of all the available cores. We also expect that the RenderScript runtime adds some overhead, due to the host code that is serial.

Figure 3 shows the results of both the reference and the RenderScript implementation while varying the image size. We found that indeed the execution time has a linear relation with the image size for both implementations. Furthermore, RenderScript does not achieve a 4x speedup, indicating that the usage of the RenderScript runtime introduces overhead. To get a better idea about the runtime overhead, we performed a second experiment where we varied the number of active cores for the RenderScript implementation with *hotplug*.

The results of this experiment are shown in Figure 4, where we normalized the RenderScript execution times with respect to the reference execution time to calculate the speedup. From this figure we can see that the image size does not impact the scalability of RenderScript significantly. The overhead of using RenderScript on a single core is 25.9%

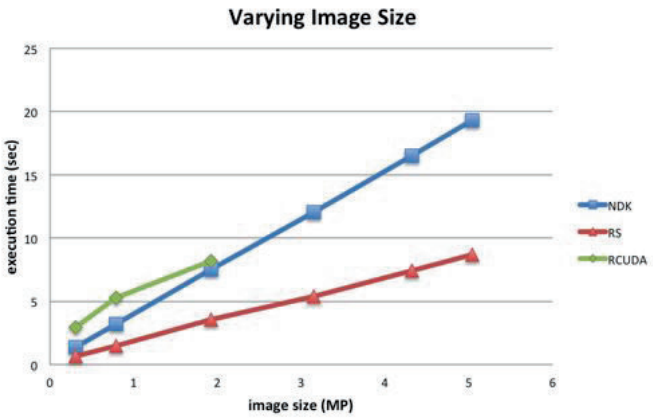


Figure 3: Execution times for RCUDA, RenderScript (RS) and the reference implementation (NDK) while varying the image sizes. The RCUDA execution times are measured without any additional latency and bandwidth constraints.

on average and increases to 45.7% on four cores. Although increasing the number of cores leads to an increase in overhead, the speedup factor increases too – up to 2.2x on four cores. This means that we have not yet reached an assymptot in the speedup graph, and although the current hardware prevents us from running the algorithm on more than four cores, adding more cores can possibly lead to even lower execution times and thus higher speedups.

We also perform the same experiment without explicitly turning on or turning off cores with hotplug, but rather letting the default governor activate cores when needed, such as would happen in real world scenarios. We find that it takes a constant time period for the governor to turn on all four cores (about 0.5 seconds). With small problem sizes (such as VGA resolution), the activation time for the other cores wastes the possible speedup severely, whereas the the activation time is hardly noticeable for an image size of 5 MP. A possible solution to improve the execution time with small problem sizes in real world scenarios is that the governor could offer an interface to applications such that they can explicitly instruct the governor to turn on multiple cores if some compute intensive job is about to start.

Now that we have seen that the RenderScript implementation improves the execution time by using multiple cores, we analyze what the impact of using multiple cores on the energy usage is. On the one hand we expect RenderScript to consume less energy, because of the shorter execution time, on the other hand we expect RenderScript to consume more energy because it uses multiple cores. If we only consider the energy usage of the quad core, we find that RenderScript's shorter execution time with a higher power draw results in energy usage equal to the NDK's longer execution time with lower power draw (see Figure 5). This is surprising given the fact that some of the energy of RenderScript is spent on overhead and one would therefore expect that RenderScript would consume more energy.

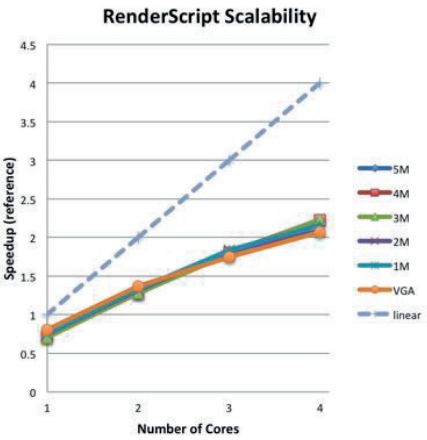


Figure 4: Although the RenderScript implementation does not reach linear speedup, adding cores improves the speedup compared to the native reference implementation.

Further analysis of the measurements reveals that the power draw of the quad core CPU does scale linear with the number of active cores, but also includes a fixed draw, and can be roughly approximated by the following formula: $P_{quadcore} = 500mW + n * 500mW$

Using more cores therefore results in better power efficiency. In the case of the RenderScript implementation however, what is won in efficiency due to the use of multiple cores is wasted on the RenderScript runtime overhead, resulting in an equal energy usage to the reference implementation.

From the experiments with RenderScript we conclude that the exposure fusion algorithm benefits from a multicore implementation, leading to a maximum speed up of 2.2x on four cores, while using an equal amount of energy on the CPU. Although the energy usage for the CPU is equal, RenderScript will likely lead to device wide energy savings, because the shorter the algorithm has to run, the shorter other components, such as the screen have to be turned on. Furthermore, for smaller size jobs the use of RenderScript will not automatically lead to good speed ups, because it takes some time for the CPU governor to switch from single core to quad cores.

5.2 Computation Offloading

In this section we turn our attention to the experiments we performed with the computation offloading implementation based on Remote CUDA. Our primary focus is how latency, bandwidth and compression level impact both execution time and energy usage.

In our first experiment we optimized the circumstances to get the lowest possible execution times for computation offloading. This means that we did not put limits on the bandwidth and latency and used compression level 1, as we show in later experiments this turns out

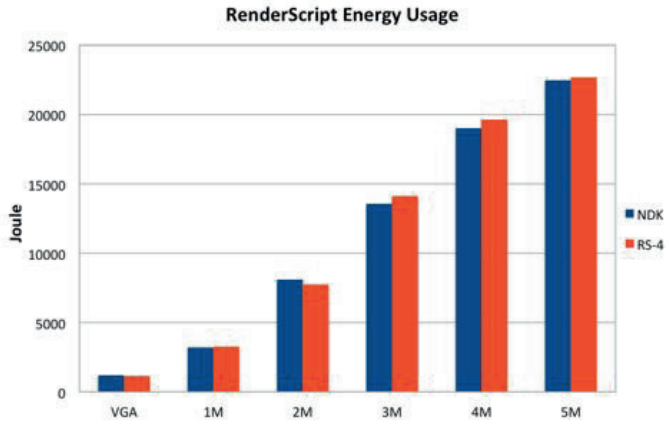


Figure 5: Energy used by RenderScript on 4 cores compared to the reference implementation.

to be the optimal compression level value. Figure 3 shows the results of this experiment. Because of GPU memory limits on our offloading laptop, which has a 512 MB Fermi based GPU, we could not run the algorithm for images with a size larger than 2 MP. Although the RCUDA implementation for all our measurements is slower than the reference implementation, we see the difference between the two implementations decreasing when the image size increases. Future experiments with a host with larger GPU memory have to prove whether this is a trend and the RCUDA implementation is faster than the NDK implementation with sufficient large images.

Because the remote GPGPU computation is much faster than the computation of the reference implementation, much of the total execution time is determined by the communication. The time spent in communicating depends on the available bandwidth and the latency. We performed a second experiment with RCUDA in which we vary both the bandwidth and the latency. The results of this experiment are shown in Figure 6. We observe that increases in the latency and decreases in the bandwidth to real world values lead to dramatic increases in execution time, well above what is acceptable for an algorithm like exposure fusion (for instance a latency well above 50 ms is common in 3G networks [HXT+10]). Therefore we can conclude that for this particular algorithm the RCUDA computation offloading framework is not a competitive alternative to on device computation. This is partially due to the abstraction level of the RCUDA framework – a low abstraction level leads to many messages, sensitive to latency – and partially due to the fact that the algorithm blows up the data, making the algorithm sensitive to bandwidth. For instance, single 1MP images, which as compressed JPEG images are typically below 200 kB, get converted to 9 MB float arrays in the algorithm. Other computation offloading frameworks that operate at a higher abstraction level (such as [CBC+10, KPKB10] that operate on the method level), could reduce the number of messages to a single response/reply and the data to JPEG compressed images.

In order to limit the impact of bandwidth on the execution time we added compression to

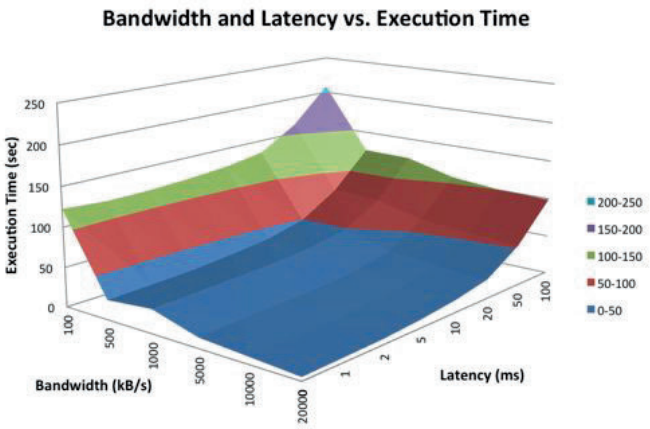


Figure 6: The impact of both bandwidth and latency on the execution time of the RCUDA implementation on a 1MP image.

RCUDA by using the zlib [ZLi] library. The zlib library supports compression levels from 0 to 9, where a higher compression level makes use of better compression techniques, at the cost of more computation. Thus with compression we can trade communication for computation. We performed an experiment where we varied the compression level and bandwidth. We expect that when there is plenty of bandwidth only simple compression will contribute to a lower execution time, whereas at low bandwidth it may be worthwhile to spend additional time on compressing to reduce the data that is sent.

Figure 7 shows the results of this experiment. We find that indeed increasing the compression level in the cases where we have a bandwidth of more than 100 kB/s only slows down the algorithm, whereas with the lowest bandwidth setting we see that an increase in compression level – beyond level 2 – leads to slightly lower execution times. However, compression level 1 is even at low bandwidth an equal choice to compression level 9, indicating that even at the lowest bandwidth that we used in our experiment, putting more effort in compressing data does not improve execution time. If we shift our focus from the execution time to the energy usage, we can see clearly that an increase in compression level, and thus an increase in computation leads to an increase in energy usage of the CPU (see Figure 8). This gives additional reason to only use simple compression. Whereas we expected that computation offloading would reduce the energy consumed by the CPU, we see that without compression RCUDA uses only 5% less energy on the CPU than our reference implementation and with compression it always uses more energy. Whereas these figures only compare energy used on the CPU rail, we should not forget that offloading computation introduces additional energy usage for communication. Since we use Ethernet over USB in our experiments, we did not include the energy usage for communication, because in real world settings a wireless variant will be used for connectivity. However, we can safely conclude that RCUDA computation offloading for the exposure fusion al-

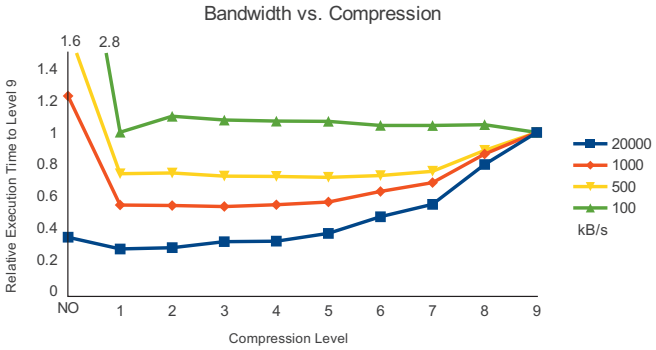


Figure 7: The relative execution time for a specific bandwidth and compression level compared to the execution time with maximum compression on a 1MP image.

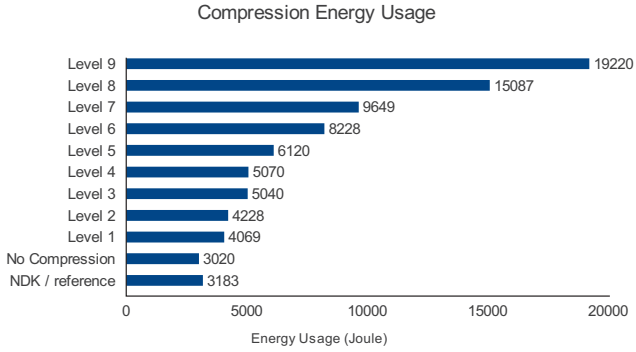


Figure 8: Energy usage at different compression levels for RCUDA with a 1MP image.

gorithm on a Tegra-3 device does not lead to better execution times nor to lower energy usage if the additional communication cost is taken into account.

6 Conclusions

In this paper we discussed and evaluated two alternative programming models to a native implementation for compute intensive tasks on mobile devices using a case study with the exposure fusion algorithm used for HDR photography. We found that using RenderScript, a multicore programming model, we can improve execution times up to 2.2 times while keeping energy usage on the CPU similar and reducing energy usage on the system as a whole. The other programming model we examined, Remote CUDA for computation offloading, did not lead to speed ups nor to energy savings, but the case study taught us several lessons that we applied in optimizing the Remote CUDA environment, such

as selecting the right TCP settings as well as improving on caching and asynchronous execution.

We also found that for short run compute intensive tasks the power of a multicore processor is not optimally used, because of the time it takes to switch from single core to quad core and therefore we recommend additional API calls for developers, such that they intentionally can turn on multiple cores just before a compute intensive task starts, instead of waiting on the processor governor to do so.

References

[CBC+10] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[CUD] NVIDIA, C Programming Best Practices Guide, CUDA Toolkit 4.2.

[DIM+10] J. Duato, F. Igual, R. Mayo, A. Peña, E. Quintana-Ortí, and F. Silla. An efficient implementation of GPU virtualization in high performance clusters. In *Euro-Par 2009–Parallel Processing Workshops*, pages 385–394. Springer, 2010.

[Eri05] M.A. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *Proc. of the USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[H+05] S. Hemminger et al. Network emulation with NetEm. In *Linux Conf Au*, pages 18–23, 2005.

[HXT+10] J. Huang, Q. Xu, B. Tiwana, Z.M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 165–178. ACM, 2010.

[KPKB10] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a Computation Offloading Framework for Smartphones. In *MobiCASE '10: Proc. of The 2nd International Conference on Mobile Computing, Applications, and Services*, 2010.

[LWX01] Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '01, pages 238–246, New York, NY, USA, 2001. ACM.

[MKVR07] T. Mertens, J. Kautz, and F. Van Reeth. Exposure fusion. In *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*, pages 382–390. IEEE, 2007.

[Nag84] J. Nagle. Congestion control in IP/TCP internetworks. *ACM SIGCOMM Computer Communication Review*, 14(4):11–17, 1984.

[Ope] OpenCV. http://opencv.willowgarage.com/wiki/.

[QZL12] Xi Qian, Guangyu Zhu, and Xiao-Feng Li. Comparison and Analysis of the Three Programming Models in Google Android. In *First Asia-Pacific Programming Languages and Compilers Workshop (APPLC)*, 2012.

[Ren] Google RenderScript. http://developer.android.com/guide/topics/renderscript.

[ZLi] ZLib. http://zlib.net/.

[ZTQ+10] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R.P. Dick, Z.M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.

Mobile Scrum

Stephan Krusche, Tobias Konsek

krusche@in.tum.de, tobias.konsek@mytum.de

Abstract: In this paper we discuss whether mobile applications can support software developers to improve their efficiency and communication. We propose the idea of Mobile Scrum, a native and integrated mobile application that supports scrum teams in their activities and fits into their environment. Team members can use it anytime and anywhere to communicate and share knowledge within the team.

Mobile Scrum provides a lightweight, yet easy and intuitive to use interface compared to existing web-based and desktop applications. With guides and templates it helps to prevent typical problems when applying Scrum. Furthermore it improves the consistency of scrum artifacts and enables easy access to important information.

1 Introduction

In 1986, Takeuchi and Nonaka introduced the idea of agile methods for product development. [TN86] They focused on the change from linear to non linear and adaptive processes to be more flexible in a ”fast-paced fiercely competitive world of new product development”. This idea was adapted to software engineering by Schwaber in 1995. [S+95] He introduced Scrum as a development process to respond to change and minimize risks and shows how Scrum can be applied in software engineering projects.

Since then, the technical preconditions have significantly changed. Mobile devices, that are becoming increasingly popular [Mob12], replace desktop computers in many situations. The access to information is possible anywhere and anytime, and the computational power of mobile devices comes close to desktop computers, facilitating productive work with these devices. The increasing integration of mobile services and applications allows the usage of context-sensitive data to provide additional features, like e.g. location-based services.

With mobile devices, human-computer interaction has become usable and has been integrated into everyday life. Ubiquitous computing describes this progress and is considered as an advancement from the old desktop paradigm. The goal is to create ”machines that fit the human environment instead of forcing humans to enter theirs.” [Wei99] Integrated tools enhance existing processes and do not ”hinder the workflow or frustrate users”. [YP04]

With these trends the question arises whether we can create mobile applications and services that improve the software engineering process. In this paper we want to investigate one case, the integration of mobility into lightweight, agile projects that apply Scrum. We describe a mobile application, called Mobile Scrum, that can improve efficiency and com-

munication within a team and integrates tightly into developers and managers workflows. We think that a mobile application can help to avoid typical mistakes that often occur when applying Scrum, especially with unexperienced teams. Maintaining communication artifacts like e.g. the product backlog in a consistent way can also be simplified with mobile tool support.

The paper is organized as follows. Section 2 describes typical problems that can occur when applying Scrum in more detail. It further explains why an integrated and native mobile application is helpful to overcome some of these problems. In the third section we show related work in the area of software engineering on mobile devices and how this paper relates to it. Section 4 clarifies why usability is our main design goal. It further shows two scenarios in which Mobile Scrum can be used to improve communication and consistency. In section 5 we summarize the advantages and describe possible future work.

2 Motivation

Some typical mistakes often occur in Scrum projects. Kniberg, a certified scrum trainer and member of the Scrum Alliance, teaches ”10 Ways to Screw Up with Scrum and XP”. [Kni08] One problem he observed in many projects, is missing information about the velocity of teams. In these cases planning needs more time and is not as efficient as possible. Estimations for the amount of user stories, that can be completed during one sprint, are not accurate. Mobile Scrum shows the velocity in burn down charts which are refreshed automatically with every change in the system. Thereby, the progress is transparent at any time and planning becomes easier.

Kniberg also observed that the product backlog is often not maintained correctly during the project which leads to inconsistencies. If the product owner does not prioritize, it is hard to decide which user stories will be implemented during the sprint. If the size of a user story is too large, the team cannot estimate its effort correctly and might not complete it in one sprint which is one principle of agile planning. [Coh06]

We think that a mobile application reduces the time of planning and supports the product owner in creating and maintaining the product backlog. For managing user stories Mobile Scrum follows the ”As a user, I want” template that ”helps the product owner prioritize”. [Coh08] This structure includes the type of user, the goal and the reason of the user story. With this approach Mobile Scrum can simplify and improve planning and provides clear insights to the roadmap and business plan that are missing in many projects.

Sutherland, one of the inventors of Scrum, describes seven ways to fail with Scrum. [Sut93] One failure point he mentions is that important information in the daily scrum meeting, i.e. status, impediments and promises, is often not communicated by each participant. Another problem is that the daily scrum meeting often takes too much time, i.e. more than the standard fifteen-minute time box, because the whole team discusses too long on issues that can also be solved in smaller groups. We think that a mobile application can guide the team to follow the principles of the daily scrum meeting more accurately.

Without using dedicated computer applications, the mentioned problems can hardly be

solved. Information can easily get lost, e.g. when a sticky note on a physical taskboard disappears or is not preserved after the end of the sprint. Even with a web based or desktop based application, it is difficult and time-consuming to store important information about tasks, impediments or backlog items in a consistent way. We observed that meeting minutes are often inserted into a tool after the meeting, which causes additional effort that can be avoided. Subjective impressions of different team members might lead to information loss, e.g. if the minute taker omits information that is important to other team members. With Mobile Scrum, notes can be collected and visualized directly during the daily scrum meeting to prevent these problems.

Another problem is that existing web based applications for Scrum (e.g. Greenhopper[1]) are not optimized to be used on mobile devices, i.e no native mobile application exists. When using a web application on a mobile device, offline usage is limited, sensors and actuators of mobile devices cannot be used and the performance of web pages is too slow for a satisfying user experience. Some existing tools are just adopted to support Scrum and do not fulfill the users expectations concerning Scrum. For instance, only a few of the investigated existing web applications include a taskboard, one of the central Scrum artifacts that should be used to visualize sprint progress. [Coh09]

3 Related work

”Software tools and environments that are designed to enhance productivity” during the software engineering process, by automating tasks or making them more enjoyable, are called computer-aided software engineering (CASE) tools. Reiss seperates CASE tools into categories like program editors, design editors or testing aids and concludes that a future category of case tools is needed that ”provide assistance in bookkeeping and managing the process”. [Rei96] Meanwhile management tools are available for web-based and desktop-based platforms. A native mobile application that integrates into the developers workflows is currently missing.

Tillman, a researcher at Microsoft who explores the *Future of Software Engineering on Mobile Devices*[2], states ”the world is experiencing a technology shift”. Mobile devices are becoming increasingly popular in work environments and present several challenges for user interface design, especially when going beyond consumer tasks. In [TMdH+12] he presents TouchDevelop, a CASE tool within a mobile application that can be used to develop mobile applications directly on the mobile device. Thereby, TouchDevelop ”embraces the form factor and input capabilities of such mobile devices”.

Overall, mobile CASE tools are quite an unexplored research topic with few publications. With Mobile Scrum we want to contribute to this research area by investigating the usefulness of collaborative issue- and task management on mobile devices to support the software engineering process.

[1] http://www.atlassian.com/software/greenhopper

[2] http://www.cs.sun.ac.za/colloquia/future-of-software-engineering-on-mobile-devices

4 Solution

Increasing mobility allows software development teams to work anywhere and anytime. This can enhance the individual freedom of team members and can help them to improve their efficiency and productivity, but it is only possible if they are able to work on their mobile device and if work artifacts are synchronized on a central server. Mobile Scrum applications synchronize data with an issue tracking server[3] as shown in figure 1 when the mobile device is online. Thus, teams benefit from a fast information distribution and improved collaboration. This would not be possible with a physical taskboard, especially if the team is not able to work in the same office. Furthermore, existing desktop or web based applications are not able to provide the same level of mobility.

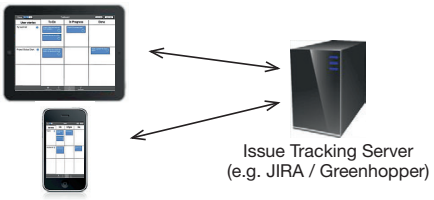


Figure 1: Collaborative support of Mobile Scrum synchronizing with an issue tracking server

There are certain challenges that need to be addressed when developing a mobile application to support Scrum. According to the Agile Manifesto [BBvB+01], software development needs to be lightweight and collaborative. Therefore usability and user experience of Mobile Scrum are decisive factors.

Nielsen defines five key attributes of a usable system: learnability, efficiency, memorability, error handling and satisfaction. [NH93] We support these aspects in the following way. Mobile Scrum integrates tightly with the main Scrum workflows and artifacts: Product backlog, sprint planning, taskboard (with daily scrum support) and sprint review. This allows the user to get familiar with the application quickly even if he only has little knowledge of Scrum (learnability). It also avoids overloading of the user interface which leads to an easy to use and productive application (efficiency). Mobile Scrum is developed as a native mobile application and provides well known standard platform controls and interactions like gestures (learnability, memorability, satisfaction). With the "As a user, I want" template for user stories, and with user guides (e.g. during the Daily Scrum Meeting), Mobile Scrum also reduces typical mistakes when performing Scrum (error handling).

Another challenge on mobile devices is the limited space compared to desktop computers. It is important to meet the users expectations with a mobile application that should be used in everydays work-life. Mapping these users expectations is described by Norman and Draper. [ND86] In Mobile Scrum, the user model represents how the user understands and applies the scrum process. The (user) interface model is influenced by the system model, i.e. the implementation of the workflows in Mobile Scrum, and by the design model that represents the understanding of the developers how Mobile Scrum should be

[3]This server is integrated into the existing environment, e.g. as plugin into JIRA / Greenhopper.

used. Testing Mobile Scrum with real users is important to determine whether design, user and interface models match with each other.

In the following, we describe two typical scenarios for Mobile Scrum. The first one focuses on supporting planned daily scrum meetings. The second one shows a typical mobile usage of the application in an unplanned situation.

4.1 Scenario: Daily scrum with a flat screen

By offering guided steps and visualization of current issues during the daily scrum, common mistakes - described in section 2 - can be prevented. At the beginning of the meeting, the scrum master Bob opens Mobile Scrum and mirrors his iPad wirelessly to a flat screen[4] as shown in figure 2.

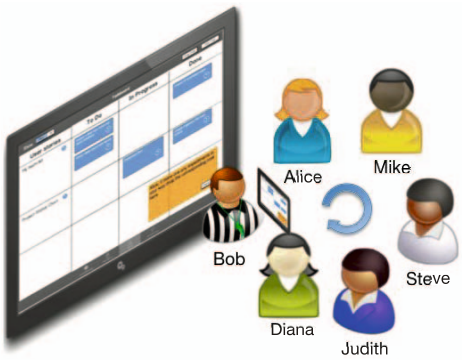


Figure 2: Daily scrum meeting with Mobile Scrum connected to a flat screen

Bob opens the taskboard view in Mobile Scrum and touches the Start Daily button. He confirms that all five developers participate in the meeting and chooses to use the guide because the team is not experienced with Scrum. He turns on the alert option to get notified when one participant takes more than three minutes. This helps the team to avoid long discussions in the fifteen minutes time-box.

The daily scrum starts and Bob passes the iPad to the first developer Alice. The taskboard adjusts automatically, to show all tasks that are assigned to Alice and all yet unassigned tasks. Alice answers the three questions presented to her at the bottom of the screen, starting with what she did since yesterday. While explaining to the team that she finished the *Dashboard View*, she adjusts the taskboard by moving the corresponding item from the *In Progress* to the *Done* column. Then she promises that she will work on the task *My Team Filter* until tomorrow and drags the task from the *To Do* to the *In Progress* column.

Finally, she reports an impediment that blocks her from solving another task. She drags the task *Query online users from Server* to the impediments area on the right bottom of

[4]This flat screen replaces the traditional physical taskboard that shows the progress of the sprint.

the screen and says ”Server hardware seems to be broken”. Mobile Scrum recognizes her voice and records the impediment. Then Alice passes the iPad to the next participant Mike, whose turn it is now. This procedure is repeated until every team member has reported his status. Then the daily scrum is automatically finished and Mobile Scrum shows the final taskboard.

After the meeting, Bob solves the technical issue reported by Alice by installing a new hardware component. Then he selects it on the taskboard and marks it as solved. Alice is notified and can continue to work on the server connection task.

4.2 Scenario: Mobile Scrum on the go

The product owner Steve is walking home from work through the park. He suddenly has an idea of a feature for the product. He opens Mobile Scrum on his smartphone. The application recognizes by using GPS information that Steve is moving and activates voice commands. Steve says ”New feature Dashboard for next release”. The application adds the new feature to the product backlog into the section for the next release and sets the title. Steve can now provide additional information. He says ”High priority” and the application adjusts the priority of the feature. The new feature is automatically synchronized to the server. The scrum master Bob, who is preparing the sprint planning meeting for tomorrow, is notified about the new feature on his mobile device.

5 Conclusion

In this paper we discussed whether mobile applications can support software development by investigating mobile scenarios in Scrum. We introduced the idea of a mobile, native and integrated application, which we call Mobile Scrum. Existing scrum applications are mostly web based and not implemented natively for mobile devices which leads to usability problems. With a mobile application, it is possible to access information anytime and anywhere. This leads to new mobile scenarios and further decreases the need for desktop computer to access important project information.

Mobile Scrum processes information and data changes automatically in the background and provides software developers an easy and consistent access to relevant information including progress and impediments. The application is designed to guide unexperienced teams through activities like daily scrum meetings step by step. This helps to avoid typical mistakes that often occur when performing Scrum.

Our goal is to implement a first prototype of Mobile Scrum in order to conduct usability tests with users applying it in their work activities. We plan to perform a case study to analyze the usage of Mobile Scrum in real working environments and we want to show that software development processes can be improved with integrated mobile applications like Mobile Scrum. Furthermore we want to investigate whether Mobile Scrum supports distributed teams in which communication problems occur more often. [SVBP07]

The mobile impact on Scrum itself is another interesting research topic. We think that discussions in small unplanned and informal meetings (e.g. at the coffee machine) can now be minuted to increase the information sharing within the team. The question arises if Mobile Scrum can be further enhanced to meet the needs of distributed mobile users. Remote daily scrums might be possible in an easier way if Mobile Scrum additionally offers video conferences in the future. It also might improve the communication of part time developers, if they cannot meet each day.

References

[BBvB+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for Agile Software Development. 2001. http://agilemanifesto.org.

[Coh06] M. Cohn. *Agile estimating and planning*. Prentice Hall, 2006.

[Coh08] M. Cohn. Advantages of the "As a user, I want" user story template, 2008. http://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template.

[Coh09] M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, 2009.

[Kni08] H. Kniberg. 10 ways to screw up with Scrum and XP. 2008.

[Mob12] MobiThinking. Global mobile statistics - Part A: Mobile subscribers; handset market share; mobile operators, 2012. http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/a.

[ND86] D.A. Norman and S.W. Draper. *User centered system design; new perspectives on human-computer interaction*. L. Erlbaum Associates Inc., 1986.

[NH93] J. Nielsen and J.A.T. Hackos. *Usability Engineering*. Academic press San Diego, 1993.

[Rei96] S Reiss. Software tools and environments. *ACM Computing Surveys*, 1996.

[S+95] K. Schwaber et al. Scrum development process. In *Proceedings of the Workshop on Business Object Design and Implementation at OOPSLA*, 1995.

[Sut93] J. Sutherland. 7 Ways to Fail with Scrum! 1993.

[SVBP07] J. Sutherland, A. Viktorov, J. Blount, and N. Puntikov. Distributed Scrum: Agile Project Management with Outsourced Development Teams. In *HICSS*, 2007.

[TMdH+12] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, and S. Burckhardt. TouchDevelop: App Development on Mobile Devices. 2012.

[TN86] H. Takeuchi and I. Nonaka. The new new product development game. *Harvard business review*, 1986.

[Wei99] Mark Weiser. The computer for the 21st century. *Sigmobile Mob. Comput. Commun.*, 1999.

[YP04] J. York and P. Pendharkar. Human–computer interaction issues for mobile computing in a variable work context. *International Journal of Human-Computer Studies*, 2004.

AndroStep: Android Storage Performance Analysis Tool

Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee and Youjip Won

Dept. of Electronics and Computer Engineering
Hanyang University, Korea
{77$smart$|$kisunglee$|$tearoses$|$insight$|$yjwon$}@hanyang.ac.kr

Abstract: The applications in Android based smartphones generate unique IO requests; however, existing IO workload generators and trace capturing tools are designed to neither generate nor capture the IO requests of Android apps. In this paper, we introduce the Android storage performance analysis Tool (AndroStep) which is specifically designed for characterizing and analyzing the behavior of the IO subsystem in Android based devices. AndroStep consists of workload generator, called Mobibench, and workload analyzer, called Mobile Storage Analyzer (MOST). Mobibench is an android App, which generates typical filesystem workloads, e.g., Random vs. Sequential and Synchronous vs. Buffered IO, as well as the most dominant workload in Android platform: SQLite insert/update and a write followed by `fsync()` call. Mobibench can also vary the number of concurrent threads to examine the storage and filesystem overhead to support concurrency, e.g., metadata updates, journal file creation/deletion. MOST capture the trace and extracts key filesystem access characteristics: access pattern with respect to file types, ratio between random vs. sequential access, ratio between buffered vs. synchronous IO, fraction of metadata accesses, etc. MOST implements reverse mapping feature (finding an inode for a given block) and retrospective reverse mapping (finding an inode for a deleted file). We explain the structure and usage of AndroStep in detail. We verify performance result of Mobibench on eight smartphone models.

1 Introduction

NAND Flash based storage device such as eMMC card [eMM11] and μ-sdcard, is the most popular storage media for Android based devices, such as smartphone, smartTV, Smartpad, etc. Android IO stack consists of DBMS, file system, IO daemon, and IO scheduler. Android 4.0.4 (ICS) uses SQLite [SQL], EXT4 [MCB$^+$07], mmqcd, and CFQ scheduler [Axb07] as DBMS, file system, IO daemon, and IO scheduler, respectively. Android application uses SQLite to maintain information in persistent manner. SQLite generates 80% of entire write operations in Android platform [LW12]. Some of characteristics of IO workload in Android are as follows: (i) 4 KB random write followed by `fsync()` and (ii) Frequent creation and deletion of small (less than 12 KB) short-lived files.

In smartphone, the storage IO is one of the key factors that governs the overall system performance [KAU]. The applications, often referred as "app", in Android based smartphones generate unique IO requests. Existing IO workload generators and trace capturing tools are designed to neither generate nor capture the IO requests of Android apps. This paper introduces the Android Storage Performance Analysis Tool (AndroStep) which is

specifically tailored for characterizing and analyzing the behavior of the IO subsystem in Android based devices. AndroStep consists of workload generator, called Mobibench and workload analyzer called, Mobile Storage Analyzer (MOST).

The remainder of the paper is organized as follows. Section 2 explains existing workload generation tools and presents analysis of Android IO workload. We introduce AndroStep, Android Storage Performance Analysis Tool, in Section 3, and Section 4 presents results of experiments with AndroStep. Section 5 concludes the paper.

2 Problem Assessment

2.1 IO characteristics of Android based Device

IO characteristics of Android based device [LW12] is different from server [HS03] or desktop [ZS99, HDV+11] IO characteristics. In order to measure the performance of Android based device properly, we need a tool that can reproduce IO behaviors of Android platform. This section investigates whether existing tools are capable of capturing the Android IO characteristics and reproducing them. This section further analyzes limitation of existing tools. Through thorough analysis of existing benchmark tools, we not only differentiate our tool from existing tools but also create a basis for implementing a benchmark tool for Android-based platform.

There are many benchmark tools available for measuring the performance of file systems or storage devices; however, existing benchmark tools, such as IOzone [ioz] or Androbench [KK12], can only measure limited amount of Android resources. In this section, we briefly explain pros and cons of IOzone and Androbench and their limitation in measuring performance of Android devices. One notable work on workload analysis of Android based platform is done by Kim et al. [KLH+11]. They investigate which system services are used by Android applications and try to allocate sufficient IO bandwidth to corresponding application via computing IO bandwidth usage model. They classified applications into three classes, bursty, time-sensitive, and plain; and showed a result of using their classification and IO management usage model in media application. However, their approach has two issues. First, they chose to model the IO characteristics of well-known system service instead of analyzing IO behavior of each application. Second, they neglected to model various scenarios within running an application, and also did not consider the fact that an application utilizes multiple system services.

2.2 Workload generation of Android based Device

IOzone is a widely used benchmark tool for measuring the performance of file system as well as IO subsystem [ioz]. There are a few issues in IOzone that makes it not suitable as a benchmark tool for Android IO subsystem. First is that IOzone does not include `fsync()` followed by a 4 KB random write operation. It is not only a frequently repeated IO operation in Android but also one of most important IO operations that journal of SQLite performs. Upon receiving a data, journal in SQLite forces to commit the data via `fsync()`. Two to three `fsync()` calls are made to the storage depending on the

journal modes of SQLite. Note that IOzone affords an option to include `fsync()` in the benchmark; however it does not simulate journal of SQLite. The option includes time of `fsync()` operation in two cases, which is after finishing the whole buffered IO and file close operation.

Second is that IOzone cannot measure performance of SQLite3, a default DBMS in Android system. Therefore, one needs to implement an Android app to measure performance of basic operations such as DB insert, update, and delete in SQLite3. Another way to measure the performance of the basic operations is to implement a shell based test tool which exploits SQLite3 APIs and cross-compiler. To build a shell based test tool, SQLite3 library must be statically linked to the application in order to run such a test tool in an Android platform. A problem in such a method is that statically linked SQLite3 library is different from shared library of the Android platform. As a result, performance acquired using the testing tool significant differs from optimized shared SQLite3 library. Thus, it is impossible to acquire reliable performance. It has to be noted that SQLite3 uses Apache license, which forbids to ask for optimized version of the open source.

The third issue concerns testing method. IOzone must run sequential write benchmark operation before performing any other benchmarks. Mandatory write benchmark in IOzone incurs significant overhead in measuring large sized IO and in repeated runs of tests.

Androbench [KK12] is a file benchmark tool that runs in Android system. What is special about Androbench is that it includes options to measure SQLite operations. Although Androbench is a simple, and yet a powerful benchmark tool to measure performance of file and SQLite operations in mobile device, there are three limitations in measuring various IO characteristics of Android based devices. First, Androbench do not allow changing synchronization options such as O_SYNC, O_DIRECT, and mmap. Only available option is O_SYNC. Second, it cannot measure performance effect of `fsync()` calls. Finally, Androbench does not support multi-threading benchmark environment.

2.3 Workload Analysis of Android based Device

Traditional IO characterization study is defined on four dimensions : IO type (Read vs. Write), IO size (KB), spatial locality (Sequential vs. Random), and temporal locality (Hot vs. Cold). All these characteristics can be analyzed by examining the block access trace. To properly understand the IO characteristics of Android Apps, one needs to acquire the four defined IO characteristics under various different contexts: subject to file types, block types, application processes, etc. For example, we need to understand how many IO writes are for metdata and how many writes are for synchronous when the IO is for journal file writes. Existing file system analysis tools does not provide the correlation information between the IO characteristics and the different file system attributes (block type, file type, and application process).

For detailed study, one has to acquire IO access characteristics, e.g., Read vs. Write, IO size; however, as far as we are aware of, existing workload capture and analysis tools are not suitable to study the details of application behavior in Android IO. It is not suitable because the tools does not identify the file type for a given block and identify the application

for a given block. To provide remedy to the issues, we implement two features in MOST.

- Identify the file type for a given block: From the logical block address captured by block access trace tool [AB07], we identify the type of file which the respective block belongs to. This process consists of two steps. First, from a logical block address, MOST identifies the inode number of a file where the logical block address belongs to. Then, from the inode number, MOST identifies the file type of the respective file.
- Identify the application for a given block: It is important not to confuse application dependent IO characteristics with file type dependent access characteristics. MOST allows to distinguish the two different IO characteristics.

3 AndroStep

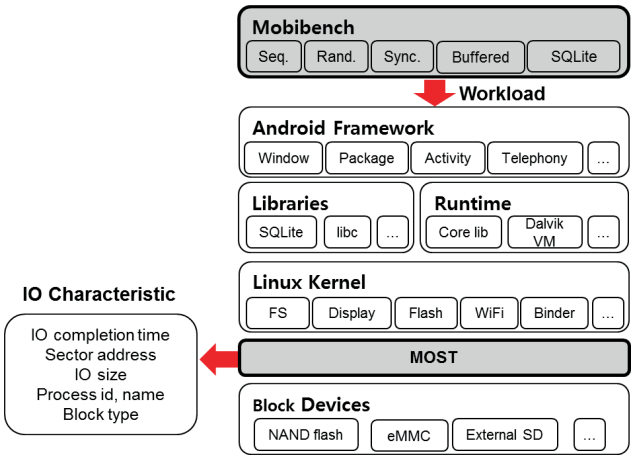


Figure 1: Structure of AndroStep (Mobibench and MOST)

Android storage performance analysis tool consists of a workload generator, Mobibench[1] [2] (Mobile Benchmark), and trace collection tool, MOST[3] (Mobile Storage Analyzer). Figure 1 illustrates structure of AndroStep, which consists of Mobibench and MOST, in the Android platform. Mobibench generates read and write IO requests which mimics the IO pattern of Android based storage device. MOST is a tool to collect IO statistics and other useful information in comprehending the IO from Android based device.

[1]Mobibench is available at google playstore.
[2]Mobibench, `https://github.com/ESOS-Lab/mobibench`
[3]Mobile Storage Analyzer, `https://github.com/ESOS-Lab/MOST`

3.1 MOBIBENCH

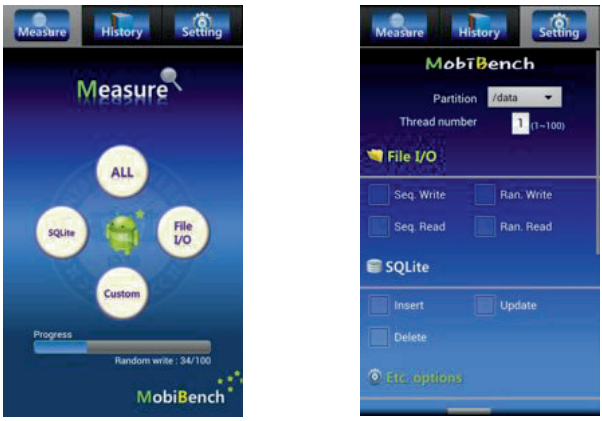
Mobibench is a benchmark tool especially designed for simulating IO characteristics of Android system; Mobibench is capable of measuring the performance of file IO and DB operations using SQLite. Mobibench is implemented in two versions, one as a shell application and the other as an Android Application (the app.) Both versions use same measurement engine written in C language. Since JAVA cannot call the app in C directly, we use JAVA Native Interface (JNI) to run the app. The shell application supports both Android device and desktop system. If the application is compiled in Android device, the application exploits SQLite shared library. Since manufacturers provide optimized SQLite library, it is possible to acquire accurate performance of SQLite operations using the app. On the other hand, on a desktop system, the application exploits SQLite static library. Since Mobibench provides unified measurement engine, it allows to test and compare the performance in diverse system.

Table 1: **Functional comparison**

	Function	Mobibench	IOzone[ioz]	Androbench[KK12]
Workload	sequential write	O	O	O
	sequential read	O	O	O
	random read	O	O	O
	random write	O	O	O
	`write()+fsync()`	O	X	X
	multi-thread	O	O	X
	SQLite operation	O	X	O
Output	throughput	O	O	O
	CPU utilization	O	O	X
	number of context switch	O	X	X
Options	exe environment	shell / app	shell	app
	separate file IO operation	O	X	X
	separate SQLite operation	O	X	X
	file sync mode	O	O	X
	SQLite journal mode	O	X	X

Table 1 illustrates comparison of three benchmarks, Mobibench, Iozone, and Androbench, on performance measurements, feedback, and miscellaneous options. Note that Mobibench combines all of the functions that are only available in Iozone or in Androbench. Mobibench provides detailed benchmark configuration options such as choice of a partition, number of threads, and workload characteristics. Once a partition and number of threads is configured, the setting is applied to all test cases. Mobibench uses one of `/data` and `/sdcard` in internal storage, or `/extSdcard` in external memory card. Differentiating the partition is important because available file system mount options for each partition are not the same. Mobibench allows configuring the number of threads for a test in order to provide similar environment as smartphones, where multiple threads execute IO and SQLite operations simultaneously.

There are two categories of workload, File IO and Database Operations. Test cases avail-



(a) measure tab. (b) setting tab.

Figure 2: Mobibench application

able in File IO category is choice of sequential and random, and read and write. Mobibench further specifies file size, IO size, and synchronous mode. The synchronous mode supports five different modes, buffered, synchronous, direct, mmap, and `write()+fsync()`. Since synchronous mode cannot be changed in file `open()` call of JAVA environment, Mobibench implements synchronous mode through C/C++ and JNI.

Database operation measures performance of insert, update, and delete in SQLite, the default DBMS in the Android system. Performance of these operations varies greatly depending on compile and PRAGMA options of SQLite. PRAGMA command is used to modify the operation of the SQLite library, which Mobibench uses to change SQLite synchronous or journal modes. According to our test, performance of SQLite varies significantly depending on the choice of synchronous and journal modes. Although modified and optimized source code of SQLite is not publically available, manufacturers provide SQLite shared library which is optimized to Android device. Mobibench uses the shared library to measure the performance of SQLite operations.

Mobibench generates three results, Throughput, CPU Utilization, and Number of Context Switches. In the case of File IO test, unit of throughput is "KB/s" for sequential operation and "IOPS" for random operation. Unit of throughput in SQLite operation is "Transaction/sec". Utilization of CPU distinguishes ACTIVE, IDLE, and IO-WAIT to understand how the test utilizes the CPU. Mobibench also counts the number of context switches to measure the context switch overheads.

One must have superuser permissions to flush buffer cache explicitly in shell version of Mobibench. Mobibench explicitly flushes buffer cache at initialization phase of the application via writing to `vm.drop_caches`.

Figure 2 illustrates user interface of the app version of Mobibench. There are three tabs in Mobibench. In Measure tab, there are four buttons, ALL, File IO, SQLite, and Custom.

File IO runs sequential or random read/write operations, and SQLite runs transactions of insert, update, and delete DB operation. ALL runs both File IO and SQLite operation and Custom runs only the tests user have selected. In Setting tab, there are customizable options which are also available in the shell version of Mobibench. When a measurement process is running Mobibench displays a progress bar to show status of a running test, and illustrates the result upon completion of the test. Mobibench shows three results, Throughput, CPU Utilization, and Number of Context Switches. Since Android applications cannot run in Superuser, Mobibench cannot flush buffer cache explicitly; instead it opens a file with `O_DIRECT` to remove the effect of buffer cache in the measurement.

3.2 MOST: Mobile Storage Analyzer

Mobile Storage Analyzer (MOST) consists of (i) a modified Linux kernel that maintains processes and file-related information for IOs; (ii) a block analyzer that enables identification of a file for a given block, and (iii) `blktrace` utility. Figure 3 provides a schematic of MOST.

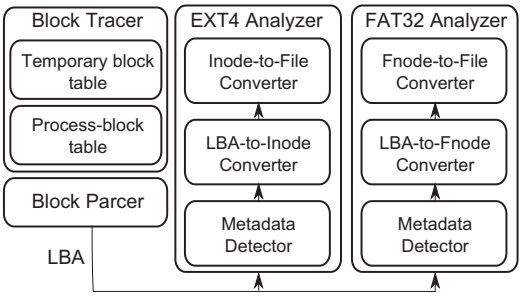


Figure 3: Mobile Storage Analyzer

Due to the layered structure of a modern IO subsystem, it is not possible to identify session-related information at the block device level. When an IO request is passed across layers, for example, from the file system to the block device layer, the session-related information (i.e., file id and process id), are lost. In order to be able to analyze the relationships among blocks, respective files, and processes, the information needs to be collected from different layers. MOST collects the IO trace at the block device driver level to deduces the file information and then it processes information for each respective block. MOST addresses three reverse-mapping issues: LBA-to-file mapping, LBA-to-process mapping, and retrospective LBA mapping.

For LBA-to-file mapping, MOST can reverse-map the disk block to the respective file where it belongs. It accepts a logical block number as an input, and generates a file name. MOST uses `debugfs` [Ts'] to reverse-map the block in the EXT4 file system, and an in-house module for the FAT32 file system.

MOST identifies the original process that issued a given IO. In Android, *mmcqd* daemon manages the mmc card device driver and is responsible for issuing all block IOs. With-

out any modification, `blktrace` reports all block IOs that are initiated by the *mmcqd* daemon, which is not the information we are interested in. We create a process-to-block mapping table in the Android Kernel. The entry of the table is <LBA, process id>. When the IO scheduler inserts the IO request into the queue, MOST inserts the <LBA, process id> information into the process-to-block mapping table. MOST references the process-to-block mapping table later in order to retrieve the process id with a given LBA.

MOST allows retrospective LBA mapping. In Android, we find that many files are short-lived and are created and rapidly deleted by SQLite. These temporary files are created by SQLite for managerial purposes, for example, creating a temporary database file for update and committing the changes to the storage device. It is very important to have proper understanding of how these files are utilized. Although they have short-life span, each files are *fsync()*ed to NAND storage, which greatly affects system performance. We need file information for a given LBA when a trace is recorded, not when posthumously analyzed. When MOST initiates analysis procedure for a given LBA, a temporary file where the block belonged might have been deleted and therefore cannot be found. To address this issue, MOST creates a file-to-block mapping table in the Android kernel. A file-to-block mapping table is an array of <LBA, file>. When the IO scheduler plugs in the LBA to the scheduler queue, MOST inserts <LBA, file> entry to file-to-block mapping table. Later, MOST references this table to obtain the file information for a given LBA. To reduce the table size, MOST inserts an <LBA, file> entry only for temporary files, that is, when the file extension is `.db-journal`, `.db-mjxxxx`, `.bak`, or `tmp`. When `blktrace` creates a log for the trace file, it consults the temporary block table to determine if the given block belongs to the temporary files that triggered the respective IO.

Table 2: **Output of Mobile Storage Analyzer**

1	IO completion time
2	Flags for read and write
3	Sector address and IO size
4	Process id and process name
5	Block type: *Metadata*, *Journal*, and *Data* block
6	File name in case of the *Data* block

MOST categorizes logical blocks into three types: *Metadata*, *Journal*, and *Data*. In the EXT4 file system, *Metadata* blocks are blocks harboring a superblock, group descriptor, data block bitmap, inode bitmap, and inode table. In the FAT32 file system, *Metadata* blocks correspond to blocks harboring a boot record and File Allocation Table (FAT). *Journal* is a journal block of the EXT4 file system. *Data* blocks are those harboring file data and directory entries. Table 2 illustrates the entry format of MOST output.

4 Experiment

We verified performance result of Mobibench against similar benchmark tools, IOzone and Androbench. We tested common features in the benchmark tool to verify accuracy of Mobibench, and show performance result of additional features of Mobibench. The results

of the study presented here are based on the Galaxy S3 [GAL] (running Android 4.0.4 with Linux Kernel 3.0.15).

4.1 Common Features

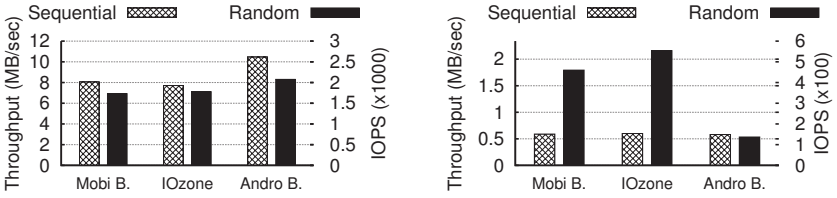


Figure 4: I/O on internal eMMC. Filesize: 512 MB, IO-size: 4 KB (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

We compared common features of three benchmarks in three different experiments. First experiment compares common features present in all benchmarks. Second experiment compares features present only in Mobibench and IOzone. Third, we compare features present only in Mobibench and Androbench. We used `/data` partition in internal eMMC. The partition is formatted in EXT4 file system.

We have validated that the result of Mobibench is accurate through measuring performance of common features available in different benchmark applications. The common feature provided in all three benchmarks are sequential and random read/write operations. We tested synchronous write and direct read performance of the three benchmarks because Androbench supports synchronous write and direct read operations only. File and IO size in the experiment is set to 512 MB and 4 KB, respectively. Figure 4(b) and Figure 4(a) shows the results from each benchmark. Mobibench and IOzone shows similar throughput and IOPS on both sequential and random operations. On the other hand, all performance result of Androbench except sequential read differs from the result of the other two benchmarks. Random write performance is regarded as critical performance measure in eMMC and NAND Flash based SSD and it is important to measure them accurately. Random write performance observed in Mobibench and IOzone shows about 10% difference; however, result of Androbench shows about 80% slower IOPS than that of IOzone.

Mobibench and IOzone supports various file open options for measuring the performance of random and sequential read/write operations. We compare the two benchmarks using six different IO operations including buffered read/write, mmap read/write, synchronous write, and direct write. File and IO size is set to 512 MB and 4 KB, respectively. We test sequential and random IO on all six modes. Result of Mobibench is not far different from the result of IOzone. On Buffered read and mmap read Mobibench shows about 25% higher sequential performance. The difference is incurred by inclusion of execution time of `fsync()` at the end of read test. IOzone includes execution time of `fsync()` in total run time of buffered and mmap read/write; however, it is unnecessary to include execution

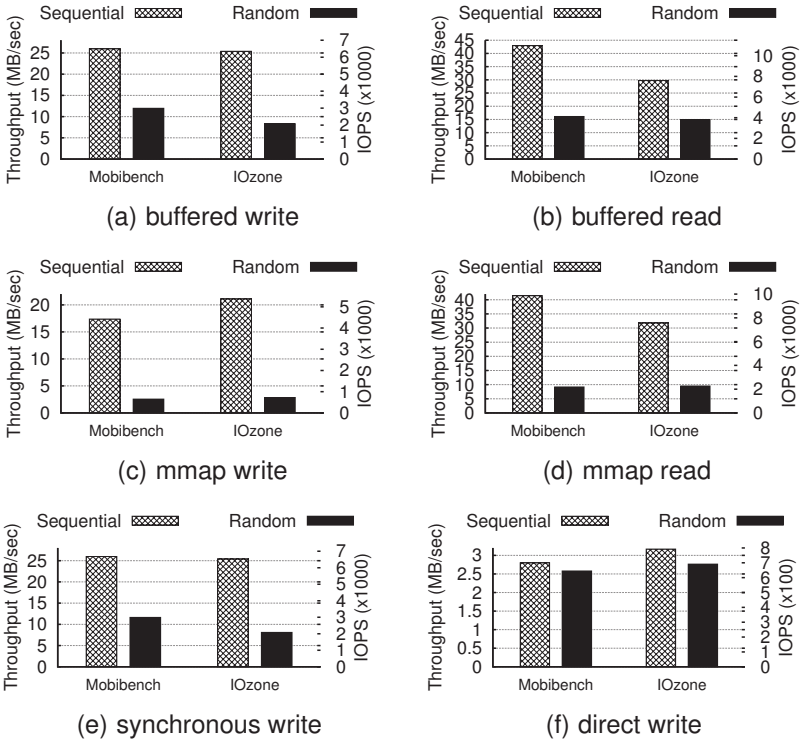


Figure 5: Mobibench vs. IOzone, IO on internal eMMC. Filesize: 512 MB, IO-size: 4 KB (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

time of `fsync()` on buffered and mmap read. We believe that buffered and mmap read result of IOzone is biased because it includes execution time of `fsync()` at the of the experiment.

Next, we measure performance of SQLite operation using Mobibench with various smartphone models and android version. We use TRUNCATE journal mode, and FULL sync mode to measure the performance and the result is average of 1000 runs. Figure 6 shows the result of the benchmark. Galaxy S3 and Galaxy Note2, which has Jelly Bean installed, show the highest SQLite performance compared to other devices. The performance seems to be affected by version of Android used in devices because same device, i.e. Galaxy S3, with difference Android version shows performance difference of more than 150%. It shows that optimization applied to Android platform, especially on SQLite, seems to produce great difference in the performance. However, it cannot be said that version of Android platform alone matters to the performance because Nexus7 and Galaxy Nexus, which also uses Jelly Bean, shows very low performance. The two devices have very low hardware specifications than Galaxy S3 or Galaxy Note. Hardware performance of a device has obvious effect on the performance.

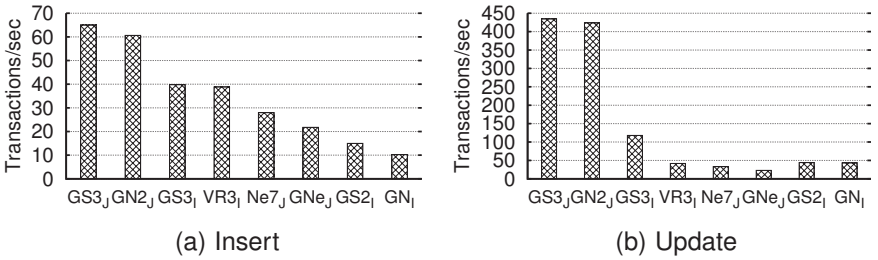


Figure 6: SQLite insert and update performance on various smartphone models. SQLite journal mode: TURNCATE. GS3: Samsung Galaxy S3, GN2: Samsung Galaxy Note2, VR3: Pentech Vega R3, Ne7: Google Nexus 7, GNe: Samsung Galaxy Nexus, GS2: Samsung Galaxy S2, GN: Samsung Galaxy Note; Subscript i and j denotes the version of Android, Ice Cream Sandwich(4.0.4) and Jelly Bean(4.1), respectively. (`/data` partition, EXT4)

4.2 New Features

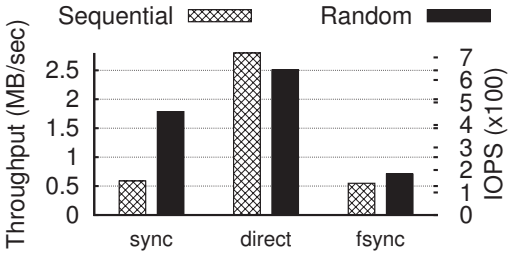


Figure 7: `write()+fsync()`, IO on internal eMMC. Filesize: 512 MB, IO-size: 4 KB (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

There are some other features that are not available in IOzone or Androbench. We devote this section to explore features available only in Mobibench. We show three different experiments in File IO, SQLite, and multithreading environment. We test File IO using `write()+fsync()`, which is dominant workload in Android system. We vary journal and sync mode of SQLite and run experiments on multithreading environment. We use the same device and partition as previous section.

Figure 7 compares the result of `write()+fsync()` against synchronous and direct write using file and IO size of 512 MB and 4 KB, respectively. IO behavior of `write()+fsync()` and synchronous write is similar on sequential IO; however in random write workload, number of IOs caused by `write()+fsync()` increase and thus it shows about twice as slower performance than synchronous write.

Figure 8 shows performance of `write()+fsync()` while increasing number of threads to 32. As number of threads increases the performance also increases because Galaxy S3 is equipped with quad core CPU; however, when there are more than 16 threads the

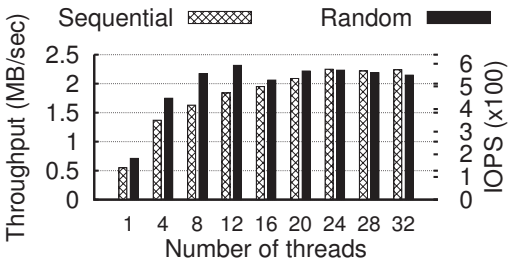


Figure 8: `write()+fsync()` with multi-thread, IO on internal eMMC. Filesize: 512 MB, IO-size: 4 KB (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

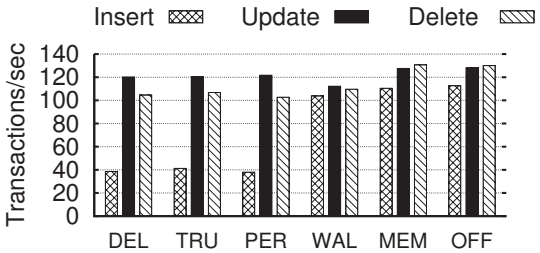


Figure 9: SQLite operation with various journal modes on internal eMMC. sync mode: FULL, SQLite version: 3.7.5 (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

performance saturates because scheduling overhead becomes the bottleneck.

Mobibench provides various options in measuring the performance of SQLite operation. Mobibench allows to change SQLite journal and sync mode. There are six Journal modes in SQLite and they are as follows: DELETE, TRUNCATE, PERSIST, WAL, MEM, and OFF. SQLite suggests that WAL Journal mode shows the best performance other than MEMORY and OFF mode [WAL]. In MEMORY mode, SQLite stores the journal information in system memory and OFF mode does not keep account of the journal. Figure 9 shows the result of average TPS of running 1000 insert, update, and delete operation. Insert in WAL mode shows about 2.5 times better performance than other Journal modes. On the other hand, update mode in WAL mode shows slightly lower performance than the other modes. Main reason behind the result is modified SQLite library in Galaxy S3. The manufacturer modified the library to use OFF mode in update operation on DELETE, TRUNCATE, and PERSISTENT mode. Using strace, we have verified that indeed all except WAL mode operates like OFF mode in update operation.

Another mode that have noteworthy effect on performance of SQLite is sync mode. There are three sync modes in SQLite which are FULL, NORMAL, and OFF mode. Number of times SQLite calls `fsync()` is determined by the mode. FULL mode calls `fsync()` on every transaction to guarantee that all the data is written to storage device. NORMAL

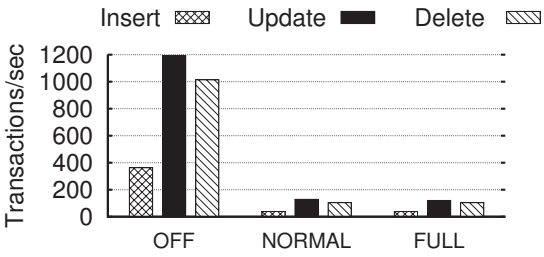


Figure 10: SQLite operation with various sync mode on internal eMMC. journal mode: DELETE (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

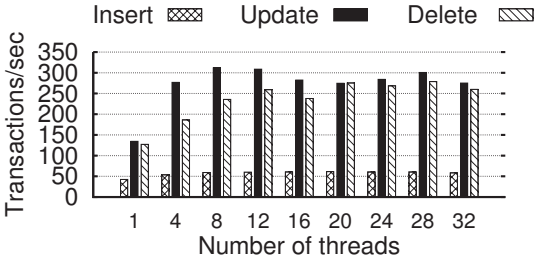


Figure 11: SQLite operation with multi-thread on internal eMMC. sync mode: FULL, journal mode: DELETE (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

and OFF mode reduces number of `fsync()` calls to produce better performance in return of sacrificing the data integrity. Figure 10 illustrates the result of three sync modes while running insert, update, and delete SQLite operations. It shows that using OFF mode speeds up greatly than using NORMAL or FULL mode in all SQLite operations.

Mobibench supports multi-threading in measuring the performance of SQLite operations as well. Figure 11 illustrates the effect of increasing number of threads up to 32. We observe no further performance gain when the number of threads is more than 12. The result is in accordance with file IO experiment and can be interpreted as scheduling overhead.

5 Conclusion

In this work, we develop a suite of software for analyzing the IO performance of the Android based device. This suite intends to generate the IO patterns which uniquely exist in the Android based storage device and which can capture the context dependent IO characteristics in various levels of the IO stack: application, file system, and block device.

6 Acknowledgements

This work is sponsored by IT R&D program MKE/KEIT. [No.10035202, Large Scale hyper-MLC SSD Technology Development].

References

[AB07] Jens Axboe and Alan D. Brunelle. Blktrace User Guide, 2007.

[Axb07] J. Axboe. CFQ IO Scheduler. In *presentation at linux. conf. au, Jan*, 2007.

[eMM11] EMBEDDED MULTI-MEDIA CARD(e-MMC), ELECTRICAL STANDARD (4.5 Device), June 2011.

[GAL] Samsung Galaxy S3. `http://www.samsung.com/ae/microsite/galaxys3/en/index.html`.

[HDV+11] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In Ted Wobber and Peter Druschel, editors, *SOSP*, pages 71–83. ACM, 2011.

[HS03] W. W. Hsu and A. J. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2):347 –372, 2003.

[ioz] IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[KAU] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. In *Proc. of the 10th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February, 2012*.

[KK12] Je-Min Kim and Jin-Soo Kim. AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices. In Sabo Sambath and Egui Zhu, editors, *Frontiers in Computer Education*, volume 133 of *Advances in Intelligent and Soft Computing*, pages 667–674. Springer Berlin Heidelberg, 2012.

[KLH+11] H. Kim, M. Lee, W. Han, K. Lee, and I. Shin. Aciom: Application characteristics-aware disk and network I/O management on Android platform. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 49–58, 2011.

[LW12] Kisung Lee and Youjip Won. Smart Layers and Dumb Result: IO Characterization of an Android-based Smartphone. In *EMSOFT 2012: In Proc. of International Conference on Embedded Software*, Oct. 7-12 2012.

[MCB+07] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proc. of the Linux Symposium, Ottawa*, 2007.

[SQL] SQLite Homepage. `http://www.sqlite.org/`.

[Ts'] Theodore Ts'o. Debugfs. `http://linux.die.net/man/8/debugfs`.

[WAL] Write-Ahead Logging. `http://www.sqlite.org/wal.html`.

[ZS99] Min Zhou and Alan Jay Smith. Analysis of personal computer workloads. In *Proc. of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS*, pages 208 –217, 1999.

Towards Automated Detection of Mobile Usability Issues

Daniel Bader
Technische Universität München
Munich, Germany
baderd@cs.tum.edu

Dennis Pagano
Technische Universität München
Munich, Germany
pagano@cs.tum.edu

Abstract: While evaluating the usability of mobile applications in the field has proven to lead to better results than in laboratory settings, in practice it is still not carried out after deployment – typically due to the required resources. In this paper we demonstrate a lightweight automated method for revealing specific usability issues of mobile applications in the field. Based on application usage data, we derive a simple heuristic which detects low discoverability by analyzing view transitions of mobile applications at runtime. We show the applicability and feasibility of our approach in a user study with a real application. Our results are promising and call for further research.

1 Introduction

Usability is a major selling point of today's software and particularly important for mobile applications which have to deal with relatively small screen estates [Hua09]. As a consequence, usability evaluation has become a mainstream activity over the last decades [CWK10]. While typical usability evaluations are carried out with a subset of all users in laboratory conditions, research has shown that evaluating usability in the field leads to significantly better results, including the identification of more issues and additional issue types [NOBP+06]. This is especially true for mobile applications, which are used in changing contexts that are often unknown before their deployment. In addition, application distribution platforms make mobile applications available to the general public and lead to more heterogeneous user audiences with different behavior and mental models. But even though it provides valuable results, usability evaluation still does not play a substantial role after deployment in practice [CKW+11], mainly because evaluating usability in the field is typically more complex and requires more resources [NOBP+06, RRH00]. As a consequence, research started to investigate to which extent usability evaluations can be automated [IH01], for instance by collecting usability relevant data like user interaction traces and automatizing its analysis [HR00]. Building on these foundations, we aim at developing lightweight and cost-effective methods for evaluating the usability of mobile applications in the field.

In this paper, we provide a proof of concept for automatically detecting *low discoverability issues* – a specific usability issue type which occurs whenever a user interface does not communicate clearly that and how the user can interact with a particular element. As a result, low discoverable user interface elements are often not found, and might even hinder

users to access particular views. Consequently, Nielsen considers low discoverability as one of the main challenges for user interfaces on touch-based mobile devices [BN].

The contribution of this paper is twofold. First, it describes a simple heuristic to detect low discoverability issues, which is derived from exploring real application usage data. Second, it shows the practical applicability of this mobile usability heuristic in the form of a lightweight framework. The framework can be integrated into mobile applications and allows for automated usability testing during application runtime.

The remainder of the paper is structured as follows. Section 2 introduces our research methodology. In the following three sections, we explore mobile application usage data (Section 3) to derive a heuristic for the detection of low discoverability issues (Section 4), and evaluate the heuristic in a user study (Section 5). Section 6 discusses the implications and limitations of our findings, while Section 7 summarizes related work. Finally, Section 8 concludes the paper and sketches our future plans.

2 Research Setting

We first summarize the questions that drive our research. Then we describe the method we used to collect and analyze application usage data and to evaluate the derived mobile usability heuristic.

2.1 Research Questions

Our goal is to investigate if low discoverability issues can be identified automatically by analyzing users' interactions. With this approach, we aim at providing groundwork towards a lightweight and cost-effective framework for mobile usability testing. Specifically, we focus on the following three research questions:

- **RQ 1: Issue indicators.** Does low discoverability appear in user interaction traces?
- **RQ 2: Detection heuristic.** Are there specific user interaction patterns which indicate the presence of a low discoverability issue?
- **RQ 3: Feasibility and applicability.** Can low discoverability issues be detected at runtime by a software framework?

2.2 Research Method

As depicted in Figure 1, our research methods consisted of three phases: a usage data collection phase, a heuristic construction phase, and an evaluation phase.

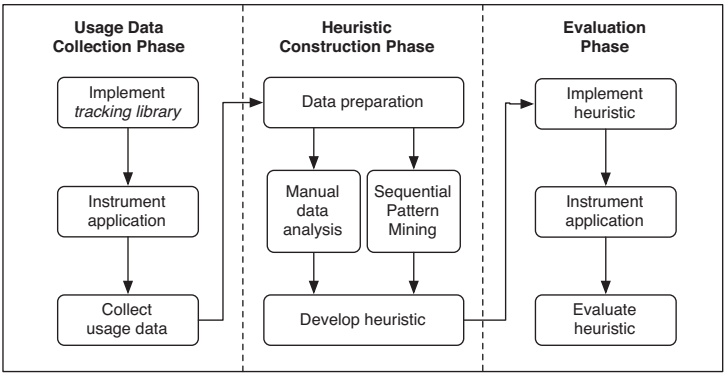


Figure 1: Research method.

In the initial *usage data collection phase*, we recorded users' interactions with a test application in order to build a corpus for further analysis. To this end, we first developed a generic tracking library which collects information about user interactions at runtime and that can be embedded into arbitrary mobile applications running on Apple's iOS[1] platform. The library records the time at which the user switches to a different view in the application together with an identifier for the new view, and logs activation and deactivation of the application. We then selected an existing mobile application for usage data collection. Apart from the obvious requirement that the application had to run on the iOS platform, we also needed access to its source code to embed the tracking library. Moreover, we had to select an application with a non-trivial navigational path, meaning that it had to contain multiple individual views and that using the application also required visiting them. Finally, we instrumented the selected application with the tracking library, and performed the actual data collection study. To this end, we asked subjects with different prior knowledge of the application to perform a sequence of tasks with it, and recorded the emerging usage data together with the ground truth about occurring low discoverability issues.

The subsequent *heuristic construction phase* consisted of four steps. After preparing the collected usage data for further analysis, we examined the obtained user interaction traces for regularities both manually and by applying a sequential pattern mining algorithm. We then used our findings to derive a heuristic classifier for low discoverability issues which works on user interaction traces.

In the *evaluation phase* we assessed the feasibility of automatically detecting low discoverability issues at runtime. As proof of concept, we implemented the detection heuristic as a software framework that can be integrated into arbitrary iOS applications. The framework constantly analyzes the occurring view transitions. Whenever the included heuristic detects a low discoverability issue, the framework notifies the user and asks if she agrees. Finally, we conducted a user study to investigate the quality of the detection heuristic. For

[1] `http://www.apple.com/ios`

this purpose, we let a different set of test users work with the instrumented test application and investigated the results in terms of true and false positives and negatives.

3 Usage Data Collection

3.1 Setup

To record how users interact with the test application we developed a *tracking library* that can be embedded into arbitrary iOS applications. The tracking library collects information about how a user navigates through the views of the application. It writes a view *presentation log* that contains a sequence of timestamped *view presentation events*. View presentation events are tracked by invoking a method whenever a view is presented to the user. Likewise, leaving and re-entering the application at runtime are encoded as special view presentation events. The data logging method requires access to the test applications source code in order to place logging statements at the relevant positions. It is self contained and works on standalone iOS devices without external connections, what enables the collection of usability data in real usage situations and locations where it is difficult to observe users directly. This benefits the results' validity because users behave more naturally when the evaluation is conducted in a familiar environment [PRS07].

We used the tracking library to collect usage data in single-participant sessions. In each session a test user interacted with a test application that was instrumented using the tracking library. All test users were asked to complete a sequence of 13 tasks. No further instructions were given during the session except when a user was stuck for more than one minute. During the sessions the test users were monitored in two ways: first, by an observer who took notes about the user's progress and second, by the integrated usability tracking library. In addition, all test users filled out a questionnaire after completing the tasks. The questionnaire contained a section for each of the 26 application views and required the test users to rate each view in the four dimensions discoverability, accidental activation, importance, and user confusion on a 5-point Likert scale (Figure 2).

After a session the recorded view presentation logs were drawn from the mobile devices for further processing. We then transformed the collected view presentation sequence into a sequence of *view visitation events*, each of which contained the name of the visited view and the *retention time*, indicating the number of seconds a user stayed in the view before leaving it. We performed this transformation in two steps. First, we corrected all timestamps by removing any time periods where the application was in the background. Second, we generated one view visitation event for each of the presented views by computing the retention times.

We used the iPhone application *MoID* as test application during usage data collection. MoID is an electronic replacement for business cards available in the iOS App Store [MoI]. The application fit the requirements from Section 2.2 because it runs on the iOS platform and its source code was accessible to us. Moreover, MoID is a sufficiently complex application consisting of 26 individual views.

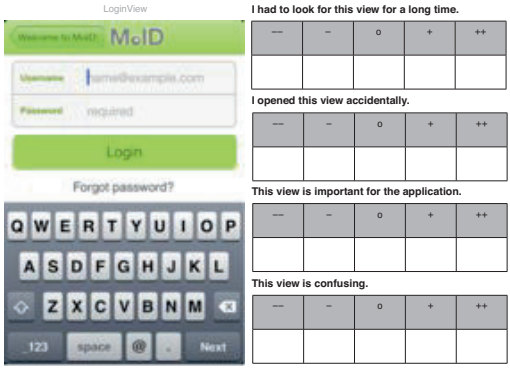


Figure 2: Sample section from data collection questionnaire.

Table 1: Overview of collected usage data.

# test users	6
smartphone owners	50%
average session length	18:09 minutes
# log file entries	581
time spent in scanning phases	722 s (12.5%)
time spent in working phases	5064 s (87.5%)

3.2 Results

An overview of the collected data is shown in Table 1. We conducted individual testing sessions with a group of 6 test subjects (3 female, 3 male). Their average previous knowledge of the test application was 2.2 on a subjective scale between 1 to 5 (1 indicating no previous knowledge, 5 indicating very high knowledge). Half of the test users owned smart phones, 1 test user owned an iPhone. During the sessions, 581 view presentation events were recorded by the usability tracking library. Each test user visited at least 16 of the 26 views of the application (61% minimum coverage) while the combined coverage of all test users was 23 out of 26 views (88% coverage).

To gain first insight into how the test users navigated through the application we analyzed the distribution of the retention times. As shown in Figure 3, there are many view visitations with short retention times and few visitations with long retention times. This observation suggests that retention times follow a power-law distribution [New05].

Next, we compared the performance of novice users to that of experienced users with previous experience with the test application. Two differences between their performances became apparent: First, novice users visited many views in order to complete the given tasks. Experienced users on the other hand exhibited almost no searching behavior, presumably since they knew what to do to solve the given tasks. Second, novice users required much more time in total to complete the given tasks than experienced users.

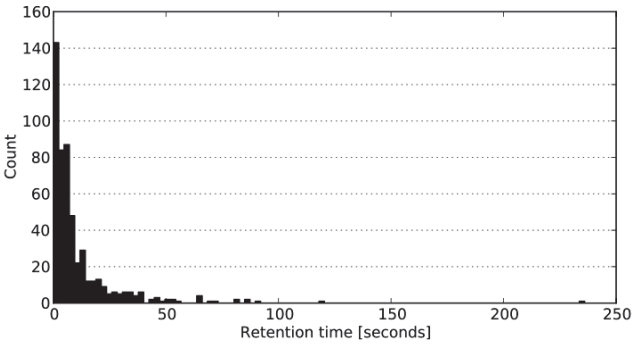


Figure 3: Ordered histogram of users' retention times.

After an initial analysis of the collected view visitation sequences we noticed that there was no connection between the observed difficulty a user had with a view and how she rated that view in the questionnaire. For example, some test users struggled with finding a view during the session but later indicated that they had no issue with the respective view in the application. We therefore decided to concentrate more on the collected observer notes for further analysis.

4 Heuristic Construction

Manual analysis of the view visitation sequences of each user revealed two kinds of behaviors that we call *scanning* and *working* phases. During *working* phases users stayed on one view for a long time – often longer than 10 seconds. These phases coincided with users working on a particular task, for example, entering information or reading descriptions. In *scanning* phases users switched rapidly between views and thus produced retention times of less than 6 seconds. Such phases appeared most often when users were looking for a particular view or function in the application in order to continue with their task.

We compared the observer notes with the view visitation sequences and investigated tasks that were difficult for the test users. We found that whenever a user had problems finding a particular view or function, the corresponding sequence typically included an overly long and misguided scanning phase. We called these misguided and unsuccessful scanning phases *problematic* scanning phases. Problematic scanning phases are characterized by three attributes in our data set. First, their view visitation events have retention times of less than 6 seconds. Second, they consist of sequences of at least 6 view visitations. Third, they contain loops, i.e., users visited at least one view multiple times during a problematic scanning phase. Figure 4 depicts how scanning phases and problematic scanning phases are identified in a user's view visitation sequence.

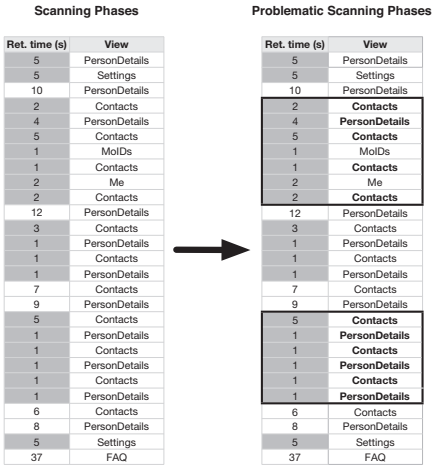


Figure 4: Example of problematic scanning phases in a view visitation sequence.

Our observer notes for each session indicated that problematic scanning phases are correlated with the occurrence of low discoverability issues. Several users failed at tasks because they missed important user interface controls in the application. Whenever a low discoverability issue occurred, users typically started to switch quickly between the views of the application. This searching behavior continued until the user could locate the user interface control or view she was looking for. We therefore hypothesized that problematic scanning phases are linked to the occurrence of low discoverability issues.

Since manual identification of problematic scanning phases is impractical for analyzing large data sets with many users in practice, we investigated if we could obtain the same results using automated techniques. To this end, we replicated the manual identification of problematic scanning phases using a sequential pattern mining algorithm by Zaki [Zak01]. For each view presentation log we generated all sequential patterns up to a maximum length of 6 items with a maximum gap value of 5. This step generated 629 patterns but this figure also includes subsequences, i.e. if the pattern list included the pattern (X, Y, Z) then it would also contain the subsequences (X, Y) and (X). After removing the superfluous subsequences we received a list of scanning phases in the input data. To identify problematic scanning phases we had to filter the resulting list of scanning phases by removing all sequences which contained no loops. Eventually, we obtained the same problematic scanning phases which we had identified manually.

We found that working and scanning phases are present in the recorded view visitation logs, assuming that problematic scanning phases consist of a sequence of view visitations with low retention times and loops. To facilitate further analysis and allow for practical use, in the following we describe a heuristic which detects these phases in sequences of view visitations. We begin by introducing the necessary terminology:

- A *view visitation* is a pair $(view, retentionTime)$.
- The *retention time* is the time in seconds the user stayed on a view. If a user enters view *A*, stays there for 23 seconds, and then navigates to view *B*, view *A*'s retention time adds to 23 seconds. This is expressed as the view transition $(A, 23)$.
- A *view visitation sequence* is an ordered list $(v_1, \ldots, v_n)$ of *view visitations*. The sequence is ordered by the natural order of events, e.g. if the view visitations A, B, and C occur one after another they are represented as the sequence (A, B, C).
- The *view visitation history* $H = (v_1, \ldots, v_N)$ is a special view visitation sequence that contains the last N view visitations. The view visitation history can be seen as a circular buffer and it is in fact implemented as such in our prototype.

We define the heuristic $ld\,(H)$ that detects low discoverability issues from a view visitation history. The occurrence of a low discoverability issue is indicated by low retention times and one or more loops within the view visitation history:

$$ld(H) := \begin{cases} \textbf{true} & \Leftrightarrow lowR\,(H) \wedge loop\,(H) \\ \textbf{false} & otherwise \end{cases} \tag{1}$$

The heuristic $ld\,(H)$ takes the view visitation history H as input and outputs a truth value that indicates if the heuristic was *triggered*, i.e. if a low discoverability usability issue is present in the input data. In the definition of ld we used two helper predicates, $lowR\,(H)$ and $loop\,(H)$. The $lowR\,(H)$ helper predicate indicates if all view visitations in the view visitation history H have retention times below a threshold value α:

$$lowR\,(H) := \begin{cases} \textbf{true} & \Leftrightarrow (\forall a_i \epsilon H : retentionTime\,(a_i) < \alpha) \\ \textbf{false} & otherwise \end{cases} \tag{2}$$

The predicate $loop\,(H)$ indicates if a view appears more than once in the view visitation history H, meaning that the user has visited a view at least twice in the last N view transitions:

$$loop(H) := \begin{cases} \textbf{true} & \Leftrightarrow (\exists a_j, a_k \epsilon H : a_j \neq a_k \wedge view\,(a_j) = view\,(a_k)) \\ \textbf{false} & otherwise \end{cases} \tag{3}$$

In our data set a history size of $N = 6$ view visitations was sufficient to distinguish problematic scanning phases from non-problematic. Likewise, a low retention time threshold of $\alpha = 6$ seconds was the most helpful when determining whether a given sequence of view visitations was a scanning or a working phase.

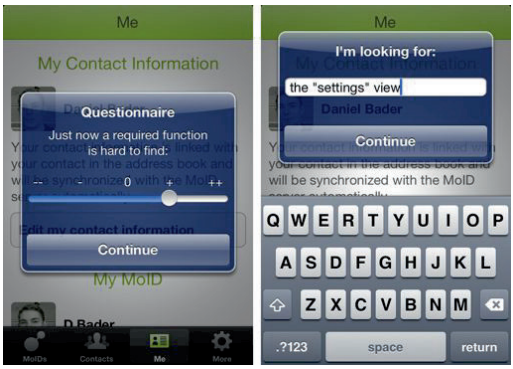


Figure 5: Screenshot of the automatic feedback survey.

5 Evaluation

We tested the heuristic in a first evaluation study with the same application and the same tasks as in the initial usage data collection study (Section 3), but carried out by a different set of users. As preparation, we implemented the heuristic as a library and included it in the test application binary. The instrumented test application analyzes the view visitation history after each view transition by the user and checks it for the occurrence of low discoverability issues. Whenever the heuristic is triggered, the application displays an *automatic feedback survey* dialog (shown in Figure 5), asking the user to rate if she currently struggles with finding a specific application element. To continue working with the application, the user has to answer the dialog by specifying her level of agreement on a 5-point Likert scale. Upon agreement, she is additionally asked to briefly specify what she was looking for. The user's response along with a timestamp and the view transition history is logged on the mobile device.

Table 2 shows an overview of the evaluation data. The evaluation study was performed with 9 test users. The level of previous knowledge with the test application is slightly lower than in the data analysis study. The average session length decreased by 33% from 18 to 12 minutes.

After each session, we determined the number of true and false positives and negatives by analyzing the accumulated log file. A *true positive* occurs when the heuristic is triggered if a low discoverability issue is present. Likewise, a *false positive* occurs when the heuristic is triggered although no low discoverability issue has been observed. In contrast, a *true negative* occurs when the heuristic is not triggered but also no low discoverability issue is present. Similarly, a *false negative* occurs when the heuristic fails to trigger, although a low discoverability issue is present. In our setting, at most $\#view\ transitions - 6$ low discoverability issues may be detected per session, which allowed us to calculate the number of true negatives. We finally estimated the number of false negatives based on our observer notes and a paper-based questionnaire after the session.

Table 2: Evaluation data overview.

# test users	9
smartphone owners	67%
avg. session length	12 minutes
# heuristic triggers	13
avg. # heuristic triggers per person	1.4
# log file entries	729

Table 3: Confusion matrix of low discoverability heuristic.

		Ground truth	
		LD^+	LD^-
Prediction	LD^+	8	5
	LD^-	3	568

Figure 6 depicts the evaluation results. It shows two bars for each test user, representing the number of times the heuristic was triggered during the evaluation session (prediction) and the number of times the test user agreed with the presence of a low discoverability issue in the automatic feedback survey (ground-truth). We found that the heuristic was triggered at least once in 6 of the 9 sessions (66%). Out of the 6 corresponding participants, 3 agreed with all predictions, 2 agreed partially, while 1 participant disagreed completely, leading to an average precision of 64% for the heuristic. If we additionally include the 3 participants who never triggered the heuristic, we arrive at a precision of 76%.

The confusion matrix shown in Table 3 illustrates the obtained evaluation results.The high number of true negatives indicates that the heuristic was typically not triggered erroneously. Furthermore, the fact that there are more false positives than false negatives means that the heuristic is more likely to trigger accidentally than to miss a real issue.

We used the confusion matrix to compute additional performance metrics for the heuristic. To do so we evaluated the heuristic like a binary classifier that predicts whether or not a low discoverability issue occurred after each view transition. Using this method we arrive at a specificity of 0.99, a sensitivity (recall) of 0.73, and a precision (confidence) of 0.61. The high specificity indicates that positive predictions by the heuristic have a high probability of being correct. Additionally, we computed the Matthews Correlation Coefficient (MCC) and Cohen's Kappa Score for the confusion matrix. Both metrics are frequently used to judge the performance of binary classifiers. The MCC and Kappa scores are both 0.66, thus indicating that our approach performs better than a classifier that randomly guesses its results.

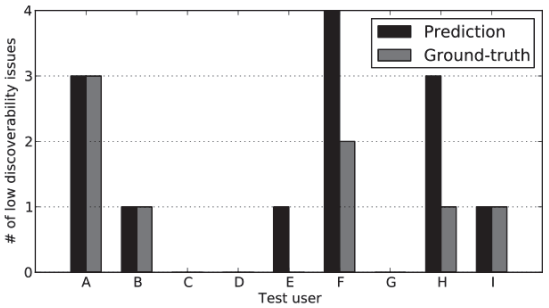


Figure 6: Evaluation results in terms of predicted issues and ground truth.

6 Discussion

6.1 Implications

Our findings represent a starting point towards testing and maintaining the usability of mobile applications in an automated way. We have shown that specific usability issues can be detected by a lightweight analysis of user interaction traces. By inspecting how users navigate through a mobile application we were able to identify patterns which might indicate low discoverability issues.

On the one hand, our findings provide foundational work for further research which should explore the potential of the proposed approach and exploit its benefits. We see three main directions. First, the applicability of user interaction trace analysis to other usability issues should be studied. To this end, researchers have to investigate which information is necessary to identify and classify additional usability issues and their respective causes. Second, researchers should collect and analyze user interaction traces from multiple users, in order to explore if specific usability issue patterns can be identified by aggregating this data. The results might provide means for an individual adjustment of usability issue detection heuristics or help to identify previously unknown issue patterns. To this end, heuristics like the one proposed in this paper could be integrated into existing software maintenance frameworks such as FastFix [PJB+12], or be made available as service working across multiple applications. Third, an automated detection of usability issues allows for a reaction at runtime. Apart from collecting data to improve usability, researchers may therefore explore ways to react *individually* to detected issues. Adaptive user interfaces might, for instance, increase button sizes if users continuously fail to select them. Moreover, contextual help might appear if and only if users are confused by a specific user interface.

On the other hand, the few requirements of our approach fit well with the restrictions of mobile devices, such as limited processing capability and power [ZA05], and ensure its practical feasibility. We have implemented the presented detection heuristic as a lightweight, application independent library, thus allowing practitioners to collect post-deployment usability data for real-world applications with little impact on battery life and performance. As a first step, developers may simply collect statistics about the usability of their applications, try to identify problematic user interfaces, and improve them. In the long run, usability data gathered from many users has to be visualized and integrated with other development data and tools.

6.2 Limitations

As with any research methodology, our choice of research methods has limitations. We therefore discuss the three main factors which might have affected the soundness of our work, and illustrate how we tried to limit them.

First, the *construct validity* of our approach might be affected if the described heuristic does not measure the occurrence of low discoverability issues but rather a related concept

such as “user confusion”. To limit this threat, we derived our heuristic from triangulated data, i.e. we aligned both user interaction traces and qualitative data gathered from users with questionnaires.

Second, the *internal validity* of our approach might be affected as study participants might have provided answers which do not completely reflect the reality, because they knew the results would be published. While this threat can never be fully eliminated in studies with real participants, we addressed it by guaranteeing our participants complete anonymity.

Finally, the applicability of our findings has to be established carefully. The main limitation to their *external validity* results from the fact that we have constructed and evaluated the proposed heuristic based on only a single application, and from the relatively small sample of test subjects. As a result, the heuristic may therefore be tied to usability issues that appear in this specific application. On the one hand, the proposed heuristic includes two parameters which allow for further adjustment to other applications and user behavior. On the other hand, our study was not designed to be largely generalizable. Rather than predicting the heuristic’s performance for all possible scenarios, its main idea is to explore the feasibility of automated usability issue detection by analyzing user interaction traces, and therefore to provide a proof of concept. Further research is necessary to validate our findings for the general case. Therefore we make the detection library available to other researchers to enable the replication of our study[2].

7 Related Work

Hilbert and Redmiles [HR00] give a comprehensive overview of techniques to extract usability-related information from user interface events of conventional desktop applications. In contrast, research on automated usability evaluation of *mobile* applications is still in an early stage. We therefore focus the related work discussion additionally on web applications, as we found this area to be the most mature. For instance, Vargas et al. [VWdR10] aim at automatically detecting usability issues in web applications. Similar to our approach, the authors propose to match user interaction sequences from web server log files against heuristics which have been specified a priori. Atterer and Schmidt [AS07] additionally show how to extend user interaction logging to include client side browser events with an AJAX based HTTP proxy. In general, conventional web applications differ from mobile applications mainly in two aspects: their user interface design and the user interaction methods. As a result, Vargas et al. perform their analysis on more data such as mouse movements, keystrokes, and accessed links. Nevertheless, our findings confirm that a similar approach is also feasible for mobile applications, and that usability issues can be detected with less interaction data available.

Most research on usability evaluation of mobile applications is concerned with supporting early design and prototyping of mobile user interfaces (e.g. [ABWD08, dSC09]). In contrast, Patern et al. [PRS07] describe a framework for the remote evaluation of applications

[2] `http://dbader.org/me13-paper`

on the Microsoft Windows CE[3] platform. Their approach is based on a task model, specified upfront by developers, which defines how users should work with the application. The framework logs users' behavior while an external software analyzes the collected data and identifies deviant user behavior, which may hint at usability issues. While our approach is conceptually similar, it works in real-time without an external analysis and without setting up an application dependent task model. As a result, our framework is more lightweight and requires less setup effort, but on the other hand might provide less flexibility. Google Analytics for Mobile [Goo11] is an extension of the Google Analytics framework for web pages, which allows developers to track user interactions within mobile applications for the Google Android and Apple iOS platforms. To use the framework, developers have to trigger specific events by calling a function in the Analytics library. A list of the triggered events is then periodically sent to a remote server for further analysis. Moreover, the framework includes a web application which summarizes and visualizes the collected data. In contrast to our work, Google Analytics does not automatically detect usability issues but instead analyzes the overall performance in the context of all users.

8 Conclusions

Evaluating the usability of mobile applications in the field has proven to lead to better results than in laboratory settings. However, in practice it is typically not carried out after the software is deployed, because the required resources are higher and such an evaluation has a higher degree of complexity. The goal of our research is to facilitate post-deployment usability testing of mobile applications by developing lightweight and cost-effective evaluation methods. In this paper we demonstrated how to detect low discoverability issues by a simple analysis of users' view transitions at runtime. We showed the applicability and feasibility of our approach in a user study with a real application. We found that our lightweight framework for automated usability testing has the potential to be integrated into other existing mobile applications.

Our results represent a starting point towards an automated evaluation framework of mobile usability in the field. In addition to this proof of concept, future research has to investigate how to detect and mitigate various different usability issues and how to benefit from usability data gathered from many users.

References

[ABWD08] F. Au, S. Baker, I. Warren, and G. Dobbie. Automated usability testing framework. *Proceedings of the 9th conference on Australasian user interface*, pages 55–64, 2008.

[AS07] R. Atterer and A. Schmidt. Tracking the Interaction of Users with AJAX Applications for Usability Testing. In *Proceedings of CHI '07*, pages 1347–1350, San Jose, CA, USA, 2007. ACM.

[3] `http://msdn.microsoft.com/en-ph/embedded`

[BN] R. Budiu and J. Nielsen. Usability of iPad Apps and Websites. http://www.nngroup.com/reports/mobile/ipad/ipad-usability_1st-edition.pdf. Retrieved October 10, 2011.

[CKW+11] P. K. Chilana, A. J. Ko, J. O. Wobbrock, T. Grossman, and G. Fitzmaurice. Post-Deployment Usability: A Survey of Current Practices. In *Proceedings of CHI '11*, pages 2243–2246, Vancouver, BC, Canada, 2011. ACM.

[CWK10] P. K. Chilana, J. O. Wobbrock, and A. J. Ko. Understanding usability practices in complex domains. *Proceedings of CHI '10*, page 2337, 2010.

[dSC09] M. de Sá and L. Carriço. An Evaluation Framework for Mobile User Interfaces. *Human-Computer Interaction–INTERACT 2009*, 2009.

[Goo11] Google Inc. Developer's Guide - Google Analytics for Mobile. http://code.google.com/mobile/analytics/docs, February 2011.

[HR00] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, December 2000.

[Hua09] K. Y. Huang. Challenges in human-computer interaction design for mobile devices. In *Proceedings of the World Congress on Engineering and Computer Science*, San Francisco, CA, USA, 2009.

[IH01] M. Ivory and M. Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4), December 2001.

[MoI] MoID GmbH. Homepage. http://www.moid.de. Retrieved October 10, 2011.

[New05] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, 2005.

[NOBP+06] C. M. Nielsen, M. Overgaard, M. Bach Pedersen, J. Stage, and S. Stenild. It's Worth the Hassle! The Added Value of Evaluating the Usability of Mobile Systems in the Field. In *Proceedings of NordiCHI'06*, pages 14–18. ACM Press, 2006.

[PJB+12] D. Pagano, M. A. Juan, A. Bagnato, T. Roehm, B. Bruegge, and W. Maalej. FastFix: Monitoring Control for Remote Software Maintenance. In *Proceedings of ICSE'12*, pages 1437–1438, Zurich, Switzerland, 2012. IEEE.

[PRS07] F. Paternò, A. Russino, and C. Santoro. Remote Evaluation of Mobile Applications. In *Proceedings of TAMODIA'07*, pages 155–169, 2007.

[RRH00] S. Rosenbaum, J. A. Rohn, and J. Humburg. A Toolkit for Strategic Usability: Results from Workshops, Panels, and Surveys. In *CHI'00*, pages 337–344. ACM, 2000.

[VWdR10] A. Vargas, H. Weffers, and H. V. da Rocha. A Method for Remote and Semi-Automatic Usability Evaluation of Web-based Applications Through Users Behavior Analysis. *Measuring Behavior*, pages 1–5, August 2010.

[ZA05] D. Zhang and B. Adipat. Challenges, Methodologies, and Issues in the Usability Testing of Mobile Applications. *Human-Computer Interaction*, 18(3):293–308, 2005.

[Zak01] M.J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning*, 42(1):31–60, 2001.

Saving Energy in Production Using Mobile Services

Christopher Ruff, Uwe Laufs, Moritz Müller, Jan Zibuschka

Fraunhofer-Institut für Arbeitswirtschaft und Organisation IAO
Nobelstraße 12
70569 Stuttgart
Christopher.Ruff@iao.fraunhofer.de
Uwe.Laufs@iao.fraunhofer.de
Moritz-Christian.Mueller@iao.fraunhofer.de
Jan.Zibuschka@iao.fraunhofer.de

Abstract: High energy costs have led to an increasing relevance of energy-efficiency over the last few years. While new equipment is mostly designed to be energy-efficient, feasible action is needed to decrease energy consumption of existing equipment on the shop-floor level. As interventions there rely on dependable information and its use at the right time and place, involvement of ICT systems and particularly mobile devices becomes evident. In our approach, a system based on a SOA back end and a mobile device-based front end was implemented as a prototype. The system uses data provided by sensors, production orders and additional metadata describing specific properties of the production systems to provide decision support and to generate recommendations for the stakeholders to realize immediate as well as longer-term energy savings.

Keywords: Energy-efficiency, Production, Architecture, Mobile Assistance

1. Introduction

Sustainable production and energy-efficient production are generally seen as the central new paradigms [JKB03] for production research within the next years. More specifically, energy-efficiency has become a more and more important aspect of sustainability, which was originally coined to describe systems allowing for an agile response to competitive challenges [JKB03].

IT Systems can contribute to realizing energy savings in several central ways: they are both needed to analyze energy inputs, which can hardly be performed manually in today's complex manufacturing processes, and to control the production machines on a fine-granular level to realize energy savings based on the analysis of the inputs [BV10].

Usage of smart mobile devices is rapidly becoming more and more prevalent in the workplace [Fo12]. While offering various means of communication, the computational power, visualization capabilities and input methods also make these devices a viable platform for the development of sophisticated business applications. Mobile assistance

systems and applications have proven helpful in various industry fields [Um09, Le07], but there have been no serious attempts to leverage their abilities in a context of improving energy efficiency in a production environment. A major challenge regarding the realization of such a system is heterogeneity, including different kinds of sensors, a wide range of components within the production systems, and third party systems such as ERP systems.

2. Requirements

There are several technical requirements that have to be addressed by the system's architecture, which were derived from a prior analysis of literature and application scenarios [LSZ11]:

Requirement I: Regarding the heterogeneous environment, that includes sensors, ERP systems and several other internal system components, the system's architecture to be able to deal with heterogeneity and interoperability aspects.

Requirement II: Because of the wide range of application areas in which the system may be used, it has to be extensible and able to integrate into other external environments with minor efforts.

Requirement III: For the hardware of the mobile assistants, several requirements have been identified: In order to realize a mobile solution, a relatively small form factor is needed, enabling the users of the assistant devices to carry the devices without much effort and without interfering with their main objectives. Specifically:

- Wireless technology is required to connect them to the SME's intranet and efficiently communicate with the services of the system's back-end.
- A high resolution display allowing the creation of an intuitive, easy to use user interface using high level widgets as well as graphs to visualize data such as the current status of the assembly components is necessary to efficiently convey complex information and data to the user.
- The assistant device should support direct (Multi-) Touch input, as the interaction with the UI (User Interface) and said high level widgets has to be easy, intuitive and barrier free. A touch based approach will allow for a direct interaction without the need to use further input devices.

There is also a set of non-technical requirements which are described in [LZS12]. Related approaches exist aiming at increasing energy-efficiency in production as well as making energy efficient systems more attractive which are also described in detail in [LZS12].

3. Realisation

3.1 Architecture Overview

The system's architecture relies on a service-oriented architecture [Ap12] and common communication standards such as the Web Service Description Language (WSDL) [WS13] and the web service protocol SOAP [SO13] are used for the communication between the services provided on the server side and the frontend on the mobile client. The system also contains a web based configuration front end which is described in detail in [LRZ12]. Regarding persistence, mass data such as frequently captured sensor data is stored in a relational data base system (RDBS) while domain specific information is managed using ontologies (see Figure 1).

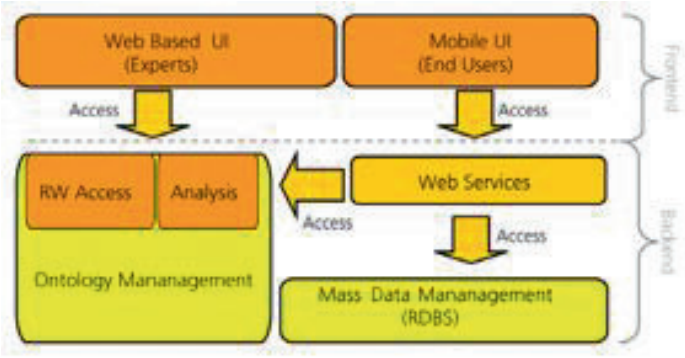


Figure 1: System Architecture Overview

The "intelligence" of the assistance system is realized on the server side as a set of web services. A detailed description regarding the backend part of the system is given in [LSZ11].

3.2 Mobile Front End

To ensure a quick response time to the proposed recommendations by the assistant system at all times and independent of the current user location, a mobile solution is preferable. This allows the system to be utilized directly in the production environment where the recommendations can be implemented directly. This also enables the responsible staff to immediately report back the changes made to the production system to the system's back-end, resulting in more up to date, accurate and reliable data. A prototype application for the mobile assistance system has been implemented and is developed further using continuous integration methods and frequent testing, using model scenarios with different users, production systems and production jobs.

The mobile front-end is realized using the software Framework MT4j [MT12]. MT4j is a java-based open source framework aimed at the creation of visually rich user interfaces which can be interacted with using novel input methods and devices, having a special

focus on multi-touch and gesture support. MT4j relies on the OpenGL API [Op12] and its mobile counterpart OpenGL ES [Op13] for rendering graphics. First developed for the desktop, it has since been ported to Google's Android platform, expanding its use onto various mobile devices. It facilitates development of graphical user interfaces by providing high-level widgets and easy to customize components while also providing ways for intuitive and customized interaction. The developer has the ability to attach individual input processors to every component, allowing for processing of input data and the recognition of the users intent. Several, input processors can be attached to a visible component simultaneously, each parsing the input data for a defined pattern. If the criteria for e.g. a flick-gesture are met, the input processor responsible for processing and recognition of that input pattern dispatches gesture events on to the presentation layer. There, gesture event listeners can be defined allowing for a custom action to be taken in case of receiving such events. The MT4j framework comes with a set of pre-defined input processors and gesture listeners covering many established (multi-) touch based interactions such as tapping, flicking, pinching or rotating. In order to cleanly separate different functionality and aspects of an application, MT4j provides a concept of having different scenes. A scene represents all the widgets, components and the corresponding input processing of a screen in an application. In our prototype, we make heavy use of this separation to facilitate readability and separation of code with regard to content. The hierarchy of the production system, the login screen or the configuration screen, for example, are each represented by a separate scene. Scenes can be pushed on an internal stack, and subsequently popped, allowing to comfortably navigating to previous scenes by "popping" them from the stack. The "Back"-Button, found on most mobile android devices usually performs this action in our application, conforming to the users' expectations.

When designing the application, several key requirements have been identified and implemented: Visualization of the production environment, Notifications, Energy Efficiency Tachometer.

Visualization of the production environment: The production sites, production systems and assembly components with all relevant data are stored in the back-end model and are fetched from the server using a web service each time the user logs into the application. The information is displayed as a list. Production components which can be navigated through by tapping onto a component to see its sub-components or details of the correspondent component:

Roles: The stakeholders of the system have been grouped into three user roles; decider, planner and machine operator.

Figure 2 View of Components

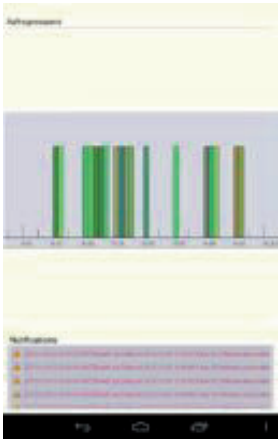


Figure 3 Production Jobs

Figure 4 Notification Details

The application provides different functionality and options for the different stakeholders and their respective roles. We address this by having the user login with their respective username and password at the start of the application. The current user role has an effect on the following aspects of the application:

Roles govern the access and permission to view production sites and production systems retrieved from the back-end model. While deciders have the ability to navigate through the entire production environment, the planners and machine operators are usually restricted to certain production sites and systems. The user role of the decider is offered an additional configuration menu where cost effectiveness of proposed measures can be calculated depending on certain, configurable variables such as aspired amortization dates or production site specific electricity costs.

The machine operator and planner are offered an up-to-date list of orders for production which can also be visualized on a timeline that the user can zoom in or out, using a pinch multi touch gesture (see Figure 3). Notifications are also tailored to the corresponding user role and can only be read if the current user role has the permission to do so.

Notifications: To notify the stakeholders about potential energy saving potential, the frontend regularly queries the backend whether new notifications are available and then transfers them to the front-end by calling the respective front end services on the server. Notifications include information about the measures to be taken in order to capitalize on the energy saving potential (see Figure 4). They conform to our role concept in such a way that every notification can only be read by a specific role or group of roles defined in the back end. The notifications can generally be grouped into two different types: Strategic recommendations and operative recommendations. Strategic recommendations mostly contain information aimed at the role of deciders and planners. The system, for example, shows the availability of alternative, more energy efficient assembly components than the currently used components. Following these recommendations, it is possible to reduce production costs in the long term. Operative recommendations are aimed at the role of machine operators. These notifications are usually more time critical

than strategic recommendations and may result in immediate energy efficiency savings. The implemented operative recommendations can contain information to temporarily shut down an assembly component or a whole production system between production jobs. The notifications are prominently displayed in a bar at the bottom of the assistant application so the recommendations are always in direct sight of the user.

Energy efficiency tachometer: As to quickly get a feel for how energy efficient the production is currently operating, a tachometer is displayed beside the production sites, systems or electric assembly components (see Figure 2). The indicator value is calculated by comparing the current operation and configuration of the production system to the optimal and worst configuration calculated in the back-end. The algorithm takes into account possible alternative assembly components as well as how well the recommendations for energy savings are followed.

4. Conclusion

We described a mobile assistance IT System that aims to support SME in order to realize energy savings in production and to thereby increase SME competitiveness by reducing energy costs. The system is designed to provide mobile ad-hoc decision support both in the planning and execution phases of production. The system's functionality employs a combination of data provided by sensors, production jobs and additional metadata describing the properties of the production systems. A service-oriented architecture is used to allow portability of the system across different manufacturing environments. In the next step, the system will be deployed to a real production system and tested in regards to functionality as well as usability and user acceptance.

Acknowledgments

We thank our colleagues from the AssiEff project for their fruitful collaboration, specifically Sebastian Schlund, Stefan Gerlach and Wolfgang Schweizer. This work was financed by Landesstiftung Baden-Württemberg under grant „AssiEff Assistenzsysteme für die auftragsbezogene, energieeffiziente Produktion“.

References

[Ap12] Apache Jena Website, http://incubator.apache.org/jena, retrieved April 23 2012

[BV10] Bunse K.; Vodicka M.: Managing Energy Efficiency. In: Manufacturing Processes–Implementing Energy Performance in Production Information Technology Systems," What Kind of Information Society? Governance, Virtuality, Surveillance, Sustainability, Resilience, 2010; pp. 260–268.

[Fo12] Forrester Research, Inc.: The Expanding Role of Mobility in the Workplace, White Paper, Feb. 2012.

[JKB03] Jovane F.; Koren Y.; Boër C.R.: Present and Future of Flexible Automation: Towards New Paradigms. In: CIRP Annals - Manufacturing Technology. 52, 2003; pp. 543–560.

[Le07] Lee R.G.; Chen K.C.; Hsiao C.C.; Tseng C.L.: A mobile care system with alert mechanism. In: IEEE Trans. Inf. Technol. Biomed., vol. 11, no. 5, Nov. 2007; pp. 507–517.

[LSZ11] Laufs U.; Schneider P.; Zibuschka J.: Design of a system for energy-efficient production in SMEs. In: ICPR 2011, 21st International Conference on Production Research. Proceedings. CD-ROM: Fraunhofer Verlag, Stuttgart, 2011; pp. 5–9.

[LRZ12] Laufs U.; Ruff C.; Zibuschka J.: Model-based support for energy efficient production in SME. In: ACM SIGCHI Symposium on Engineering Interactive Computing Systems. IT University of Copenhagen, 2012; vol. 4.

[LZS12] Laufs U.; Zibuschka J.; Schneider P.: Decision support for energy efficient production in SME. In: Mobility in a globalised world: University of Bamberg Press, Bamberg, 2012, pp. 135–144.

[MT12] MT4j Website, http://www.mt4j.org, visited: April 23th, 2012

[Op12] OpenGL Website, http://www.opengl.org/, retrieved December 27 2012,

[Op13] OpenGL ES Website, http://www.khronos.org/opengles/, retrieved January 5 2013.

[SO13] SOAP Specifications Website, http://www.w3.org/TR/soap/, retrieved January 5 2013.

[Um09] Umblia T.; Hein A.; Bruder I.; Karopka T.: MARIKA: A Mobile Assistance System for Supporting Home Care. In: MobiHealthInf 2009 - 1st International Workshop on Mobilizing Health Information to Support Healthcare-related Knowledge Work, Porto, Portugal, 2009.

[WS13] Web Service Definition Language Website, http://www.w3.org/TR/wsdl, retrieved January 5 2013.

Evaluation of cross-platform frameworks for mobile applications

Andreas Sommer, Stephan Krusche

andreas.sommer@in.tum.de, krusche@in.tum.de

Abstract: With today's ubiquity of smartphones, tablets and other mobile devices, more and more businesses develop and use mobile applications. Multiple platforms including Android and iOS form the market of mobile devices, so it can be important to be able to deliver software for more than one platform. Vendor-supported SDKs are feature-rich but incompatible with other platforms. We compared a number of cross-platform frameworks for mostly platform-independent development on mobile platforms, by evaluating them in several categories and weighing them against native SDKs. We found out that cross-platform solutions can be recommended in general, but they are still limited if high requirements apply regarding performance, usability or native user experience.

1 Introduction

The increasing demand for mobile applications in many areas [Syb11] presents developers and companies with several problems: implementing an application for multiple mobile platforms, like Android and iOS, may require high effort in terms of development time, resources, maintenance and possibly licenses, tools and deployment. One influencing factor is the fragmentation of the market of mobile platforms, i.e. smartphones and tablets. Android and iOS lead the sales with a high market share [Int12], while other platforms are less prevalent but might become more popular in the future (e.g. BlackBerry 10).

Using vendor-provided SDKs in order to implement an application for each desired platform results in highly responsive applications with a native look and feel, but has the downside of different, often incompatible programming languages, libraries, and user interface guidelines or designs. If instead an application is implemented with a common code base and possibly a single programming language or a small set of technologies, the compatibility issues are reduced, and also potentially the time required to implement and deploy the application on all desired platforms. Previous research regarding cost savings and other advantages exist in other areas of software reuse [A+09] [MC08], but not for all types of cross-platform solutions such as application frameworks. There seems to be no in-depth research available on the quantification of savings with existing cross-platform solutions, but *VisionMobile*'s developer survey of 2012 [Vis12] states that typically, more than 55% of production costs pertain to development and debugging, hinting that great savings could be achieved regarding invested time and resources for development.

Different cross-platform approaches emerged within the last years. In this paper, we describe which types of cross-platform solutions exist and evaluate three frameworks versus two native SDKs in criteria pertaining to functionality, usability features, performance and other categories. We focus on criteria and frameworks suited for developing business applications, i.e. excluding games in particular. Previous research in this area is sparse and often focuses on comparing a set of existing frameworks. We would like to conclude whether these solutions are useful at all and in which cases they can be recommended. The problem of supporting multiple platforms is not new but rather the reappearance of the same, 20-year old problem for PC platforms[1]. With the ubiquity of mobile devices and fragmentation of platforms, it makes sense for developers to target multiple platforms and thus for framework vendors to provide appropriate solutions [A+10].

In the following, we present existing types of cross-platform frameworks and the solutions selected for comparison (ch. 2). Chapter 3 details the methodology of the evaluation. We then list and explain the results for each category of criteria (ch. 4) and conclude about the advantages and problems of the solutions as compared with a native approach, and state in which cases the cross-platform approach is useful and ready for production (ch. 5).

2 Cross-platform frameworks

2.1 Definitions

Platforms are a combination of hardware (system plus any possibly built-in sensors or actuators), operating system, vendor-provided software SDKs and standard libraries [BH06], which together offer the base for building software for that platform.

A *framework* allows the reuse of a usually predefined application architecture and provides components that help in quickly setting up an application [Pas02]. Thus, a cross-platform framework allows to reuse parts of the application source code for multiple platforms and may additionally offer building blocks such as an architecture style (e.g. MVC[2]), user interface API or other functionality that is not specific to a single platform. Such frameworks can also expose APIs for platform-specific features.

Mobile applications exploit features specific to mobile devices and platforms, for instance a camera, accelerometer or fine-grained location retrieval. Pure web applications running in a mobile browser do not yet have access to all device features through a JavaScript interface [HHM12] and hence do not fall into the category of mobile applications.

[1] Windows, Unix, Mac OS, etc.
[2] Model, View, Controller

2.2 Types of cross-platform solutions

Pure web application frameworks Even though web applications without extensions were not considered as mentioned above, it should be noted that several frameworks exist which primarily rely on HTML as user interface technology and thus can be combined with pure web application frameworks. For example, *jQuery Mobile* [jF12] provides an architecture for web applications, including predefined user interface components[3], page loading and transitions. Other frameworks even contain custom styles for mobile platforms, e.g. components that look similar to iOS native user interface components.

Partially based on web technology Frameworks that primarily use HTML and related technologies to display the user interface, but additionally offer an API for calling device-specific functionality, belong to this category. Existing frameworks employ JavaScript to allow access to native functions, bridging between JavaScript function calls and native APIs (e.g. functions of the vendor-provided SDK for the underlying platform). However, there are frameworks that support calling native functionality but do not provide the means to implement user interfaces. For instance, *PhoneGap* [Ado12] offers many native functions but only supports using the platform's web view component[4] in order to display user interfaces – no UI functionality is included. Such frameworks can be combined with UI-focused frameworks, for example the web application frameworks mentioned above.

Compiled or interpreted code Unlike the previous category, frameworks of this type do not mainly use web technologies, but usually incorporate a different kind of user interface API and a single programming language which is then used across all supported platforms. For instance, the *Titanium* framework [App12] packages applications with a standalone JavaScript interpreter. Web-related APIs like DOM (Document Object Model) manipulation are not included and instead, Titanium offers its own API for generating a user interface from code, and for other purposes like network, device and file access.

Since these frameworks often support native user interface components, differences between the platforms' typical UI design must be considered. Therefore, different framework architectures can be found. Titanium offers an abstract interface for common components like buttons, but specialized APIs for platform-specific ones (e.g. iOS toolbar). Other frameworks only offer a programming language for platform-independent code, while requiring user interfaces to be developed separately for each mobile platform.

Other types A number of existing cross-platform solutions use approaches that differ from the above categories. For example, the *XMLVM* project has the goal of translating programming languages into each other. One example utilization is the cross-compilation of an Android application to a native iPhone application by translating its source code and adding a compatibility library that mimics the Android API on the iOS platform [XML12].

[3] Buttons, lists, text labels and fields, etc.

[4] Component supporting web browsing functionality, including rendering of HTML, JavaScript execution etc.

2.3 Compared frameworks

We conducted a case study comparing three cross-platform solutions with two native SDKs. The evaluation methodology is explained in the next chapter. In the following, the tested cross-platform frameworks and their architecture are outlined.

Titanium *Titanium* [App12] is a framework by *Appcelerator Inc.*, supporting Android and iOS with the latest version 2.0 at the time of evaluation. It leverages JavaScript as main development language, but does not primarily use HTML or other web technologies. HTML display using an embeddable web view component is supported, however. The architecture consists of two major parts: an application is bundled together with a standalone JavaScript interpreter to execute the application code, and the Titanium library that provides APIs for device functionality like sensors, file system access, native UI components and others, abstracting away many differences between the platforms. An application can use the Titanium library from anywhere in the source code. Calls to methods or properties of the global object *Ti* are passed to the native implementation for the respective platform. This bridging facility also allows own native extensions (platform-specific native code) to be added and used in a project from JavaScript code.

Rhodes The framework *Rhodes* [Mot12] was developed by the company *Rhomobile* which was acquired by *Motorola Solutions* in October 2011. The latest version at the time of evaluation was 3.3.2, supporting the 5 platforms Android, iOS, BlackBerry, Windows Mobile and Windows Phone 7.

The client-server model is used as basic architecture. The user interface of an application is shown entirely by a web view component that renders HTML content and evaluates JavaScript code. The application's backend contains models and controllers (based on the MVC architecture) written in the *Ruby* programming language. Views are written using HTML and a template language[5]. Any controller action can be accessed from the HTML-based views by accessing the respective URL. The framework includes a simple web server that is embedded in applications and runs locally on the device. This web server handles the translation of URL accesses to method calls in a controller and returns the view that is created as result. Controllers do not have to return a new view, but can also just run a task in the background without rendering an HTML page. They can directly interact with the user interface by sending JavaScript code which is executed by the web view.

Device capabilities can be accessed through the Rhodes library, and thus only from backend code (controllers written in Ruby, not from views[6]). A reduced Ruby standard library is included with the Ruby virtual machine that is bundled with an application. As a result, common functionality such as file access does not require the use of a separate API. Additionally, Rhodes provides modules for accessing, storing and synchronizing data.

[5] *ERB* (*Embedded Ruby*), a templating system that is also used in web frameworks such as *Ruby on Rails*

[6] In a subsequent version, it seems that some functionality can also be accessed in HTML and JavaScript.

PhoneGap and Sencha Touch *PhoneGap* [Ado12] is a framework that does not provide any UI generation functionality, but only the capability for displaying web technology based content. Therefore, we decided to combine PhoneGap with the web application and UI framework *Sencha Touch* [Sen12] because usability features are part of the evaluation criteria. The evaluation results are valid for this combination of frameworks and may vary if another web framework is chosen.

PhoneGap was initially developed by the company *Nitobi* which was acquired by *Adobe Systems* in 2011. As of the evaluated version 1.9.0, PhoneGap supports 7 mobile platforms: Android, iOS, BlackBerry, webOS, Windows Phone 7, Symbian and Bada. It allows development of mobile applications using web technologies by providing an interface to a web view component and tools to create platform-specific project files and initial source code that shows the web view. Additionally, PhoneGap provides APIs for device and platform functionality through a JavaScript-to-native bridge and native code can call back JavaScript functions inside the web view as well. This bridge is implemented differently on each platform. Developers can extend PhoneGap with native plugins by implementing a simple interface which also differs across the platforms. No user interface functionality of any sort is included, so PhoneGap can be seen as a wrapper for mobile applications that use HTML and other web technologies for displaying their user interface.

Features that are not natively supported on a platform are mimicked by PhoneGap's own implementation. For instance, if the *W3C Web SQL Database* specification is not implemented by a platform, PhoneGap's own implementation is used. Close adherence to such standards enables developers to use the same interface on all platforms.

3 Comparison

3.1 Methodology

The decision for the three selected cross-platform solutions was mostly based on active development, stability and popularity of the frameworks in order to gain an insight on solutions that are mostly ready for production use. We evaluated all solutions in five categories which are described below. Each category has a weight between 1 and 5 points (1=least important, 5=most important) which is multiplied with the score in a category. The values are summed up in order to gain the final result. Single scores are also in the range 1 to 5 (1=not fulfilled, 2=poorly fulfilled, 3=basic expectations met, 4=more than fulfilled, 5=all expectations completely fulfilled or exceeded). Since many criteria require evaluation of a framework's features from a developer's point of view, we based the comparison on the implementation of a well-defined sample business application with each solution. Criteria that are not related to development, such as license costs, were also considered to a certain extent. The main focus, however, was the usefulness of cross-platform solutions for developers and advantages for small to medium sized companies that develop mobile applications. The same methodology was applied to both

the Android and iOS native SDKs[7] so that comparable results could be achieved and potential advantages and pitfalls of both the cross-platform and native approaches would become clear during evaluation.

3.2 Sample application

The sample business application "MobiPrint" was invented as mobile app to support customers of a contrived chain of photo printing stores. Its main purpose is to order printed photographs over the Internet using a mobile device. Customers gain the convenience of sending pictures from their mobile device, without having to bring a physical medium to the store. The store itself saves time because pictures come over a common channel, and it is not necessary anymore to extract files from different media types (USB sticks, CDs, SD cards). Amongst other points, development of this application mainly tests

- **Integration with web services:** A publicly accessible, HTTP-based web service and its exposed actions are predefined and must be used by the sample application.
- **Usage of device features:** For example, the current location is used to retrieve information about nearby stores.
- **Additional data sources on the device:** In order to read and select pictures for upload to the web service (e.g. access to photos or file system).
- **Usability features:** Mockup designs for different screens were given and the UI had to be implemented from scratch with each of the five competitors. Thus, features and problems with e.g. layouting could be evaluated in each implementation.

3.3 Categories

The three cross-platform solutions and two native SDKs were evaluated in five categories which are roughly based on the *FURPS+* model [BD09][8], but adapted to describe the requirements of frameworks instead of finished software. The adapted model shall describe how well a framework supports the various categories when using it to develop software products, e.g. "features to improve usability" instead of "actual usability". FURPS+ was chosen because it concisely categorizes different types of software requirements.

Functionality This category typically comprises all functional requirements of a software product. For cross-platform frameworks, the available variety and quality of features

[7] Android SDK v10 (for Android 2.3), iOS 5.1 SDK

[8] Describes software requirements in functionality, usability, reliability, performance, supportability and other categories

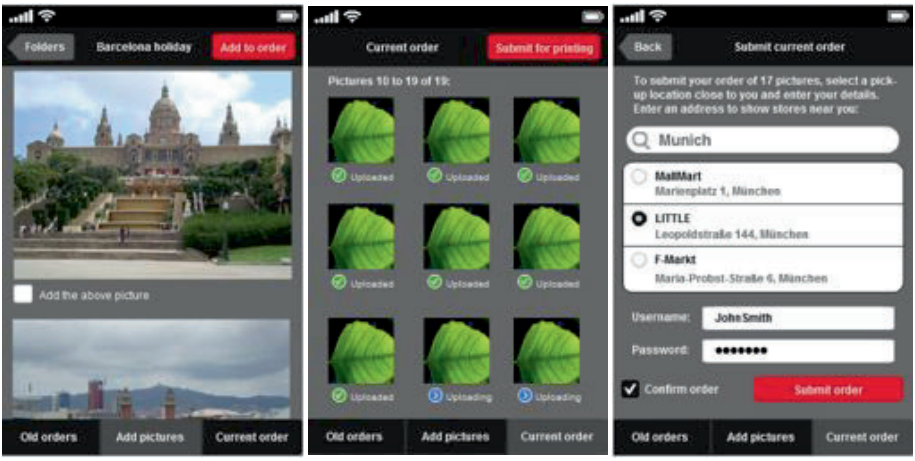


Figure 1: Mockups for the sample application screens

are considered, which includes: geolocation and network access, a persistence layer or database access, hardware sensors and actuators, buttons (hardware buttons, or software buttons provided as a persistent element of the operating system's user interface[9]), application lifecycle (events such as start, pause, resume; ability to run a background service or similar) and access to data provided by other applications or the operating system (e.g. contacts). Extensibility positively affects the score, for example if natively running code or access to features of a platform's native SDK is supported.

Usability features This category answers the question: how well does a framework support features and concepts that can help improve usability of the developed application? This includes features offered for creating a user interface: typical elements like buttons, lists, a tab or navigation bar, but also more mobile-specific elements like a paging component for switching between several pages by touch gestures.

The support for touch gestures is important because gestures (potentially platform-specific) improve learnability and satisfaction by allowing users to achieve certain actions quicker and in a familiar way (e.g. "pull to refresh"[10]). Another criterion is customizability, meaning which attributes of UI components can be changed and to what extent. This includes colors, transparency, font type and size or shapes. Corporate identity, for example, may prescribe a certain color scheme or font type. The better an interface can be accommodated to the intended design, the better its usability can potentially be – or in other words, if the necessary customizations and features are not available, some usability features (like gestures and UI patterns based on them) cannot be implemented, or only with high effort.

[9]Fixed button bar as introduced in Android 3.0, for example.

[10]If a list component is scrolled to the very top, and the user shortly swipes down with the finger, an application can recognize this gesture and e.g. refresh the list.

The incorporated points of comparison include UI functionality, i.e. basic and advanced features and components, customizability, gesture support and the ability to use native UI components, e.g. the iOS navigation bar. Moreover, the easiness of applying platform-specific looks and the overall subjective UI impression are taken into consideration.

Developer support This category aims at the ease of use that a framework offers to a developer. Documentation completeness and quality plays an important role, but also the development tools provided. Frameworks based on web technology may allow testing a mobile application in a browser or simulator, which would give developers the advantage of modern debugging tools built into browsers[11]. Other frameworks may provide their own simulator or support for using an existing simulator (e.g. iPhone simulator) to test applications. Some frameworks may restrict developer tools for common tasks (build, framework update, etc.), such as a certain IDE. Restrictions of this type may hinder the workflow of developers or existing automatic build setups. Overall, the following points are considered for the developer support category: documentation, debuggability and testability, tool restrictions and learning effort (e.g. number of technologies involved).

Reliability & Performance These categories are defined separately in the FURPS+ model – for simplicity, they are combined here. A reliable application is able to "perform its required functions under stated conditions for a specified period of time" [IEE90]. This is particularly important for mobile applications: since they typically run with restricted resources (e.g. available memory, process paused while application is in background), applications should run without crashes even if certain operations need a high amount of memory for a short time. Performance – specifically in the area of mobile applications – is mainly concerned with short response times for user interactions. Thus, an example of a poorly performing application feature could be a user interface that takes several seconds to switch between screens. Of course, general performance criteria for software apply, too, i.e. any measurable attribute that affects speed, waiting times or accuracy.

Performance was evaluated on two older test devices, the *Huawei Ideos X3*[12] and *iPhone 3GS*[13], in order to see how well the sample application implementations perform on low-end devices. For this category, the sample application's stability and size of the packaged/installed application are considered, as well as UI and application performance.

Deployment, Supportability, Costs Supportability is determined by the "ease of changes to the system after deployment" [BD09]. App stores are an important point to consider, in particular regarding the framework's support for packaging and its compatibility with restrictions and submission guidelines of the stores. Supportability also includes the portability to other systems – in this case, other mobile platforms. Experiences porting the sample business application from Android to iOS will be documented

[11] For instance the WebKit development tools that can display the HTML document tree, CSS (Cascading Style Sheets) rules, loading times, a JavaScript console and other helpful utilities

[12] Android 2.3.3, 600 MHz, 256 MB RAM

[13] Tested with iOS 5.1.1, 600 MHz, 256 MB RAM

in the evaluation. Portability for the native SDKs is rated low because native applications cannot be ported easily. This criterion is not weighted higher or regarded as separate category because there is little research available on quantitative differences in development time and costs when porting applications. Furthermore, the activity of framework development and the viability of vendors are important, especially if a business wants to use a certain framework in their long-term mobile strategy.

This category comprises a variety of criteria: a framework's supported mobile platforms, compatibility with app store guidelines, additional dependencies (e.g. a runtime environment has to be installed separately to run an application) and built-in support for internationalization, i.e. translations and localization (formatting of dates, currency values and other locale-dependent settings). We also considered support for simplified or automatic builds, for example through external build services or included tools, and checked how actively the frameworks are being developed. Regarding costs, we evaluated the direct costs for using a framework and additional (professional) support options and costs.

3.4 Category weights

Weights describing the importance of a category from the point of view of a small development company are assigned as follows (1=least important, 5=most important):

Functionality: 4 Certain built-in features and support for hardware sensors and actuators can be very important depending on the type of application. Extensibility, e.g. being able to add code that uses native functionality or to use third-party libraries, is even more important.

Usability features: 3 Usability greatly depends on a good application design. However, in case a framework does not provide extended features such as gesture recognition, some features of the finished application are harder to implement.

Developer support: 2 Tool restrictions and debuggability play a smaller role with mobile applications because they are typically less complex or require less development time than comparable desktop applications, and as such the need for helpful developer tools is not as important.

Reliability & Performance: 3 Performance is a critical point for customer satisfaction because slow loading times may prevent users from ongoing use of an application. Nevertheless, new technologies such as multi-core processors are already emerging and will help mitigate performance issues. Also, cross-platform frameworks are relatively new and still have the potential to progress in terms of performance. Together with reliability, a common and surely important aspect of software, this category is considered of medium importance overall.

Deployment, Supportability, Costs: 5 Costs and time-to-market are key factors for the success of an application. The typically short development cycle of mobile applications requires fast delivery of new versions, which can be simplified by automatic

build services or built-in packaging or deployment functionality. This category also comprises the fulfillment of app store rules, support for different platforms and how actively a framework is developed. Costs of a framework and available support plans are also evaluated. Therefore this category is most important because it includes factors that influence the cost, ease and speed of deployment in the long term.

4 Results

Table 1 summarizes the results of the evaluation. The scores show that the native SDKs win most categories. However, the actual differences in each category need to be considered.

Category	*Weight*	*Titanium*	*Rhodes*	*PhoneGap and Sencha Touch*	*Android SDK*	*iOS SDK*
Functionality	4	3	2	3	**4**	**4**
Usability features	3	4	3	3	**5**	**5**
Developer support	2	**4**	3	**4**	**4**	**4**
Reliability & Performance	3	3	2	2	4	**5**
Deployment, Supportability, Costs	5	**4**	**4**	**4**	**4**	3
Rounded final score:		**3.6**	**2.9**	**3.2**	**4.2**	**4.1**

Table 1: Overview of evaluation scores for all frameworks

4.1 Functionality

All three cross-platform frameworks support basic functionality like lifecycle events (except Rhodes), access to built-in hardware, intercepting button presses, or network, file and data access. In order to support all functions that the native SDKs offer, a framework would need many high-level abstractions or has to make compromises – Titanium supports some extra features of both Android and iOS without exposing them as platform-independent interface. The number of developers actively involved in the cross-platform frameworks is supposedly small in comparison to the native SDKs that are supported by large companies. Therefore, functionality of the frameworks is first of all restricted to what their developers can implement and support. PhoneGap takes a good approach here, in that only portable functionality is supported in the core API, whereas other features are available in separate plugins that are written in native code and thus can utilize all functions of the respective platform.

4.2 Usability features

The score difference in this category is mostly due to the support for native components or native-like styling capabilities. Titanium uses native components, but both Rhodes and PhoneGap are based on HTML, so achieving a fully native experience is cumbersome. Also, the native SDKs provide high customizability and well-designed UI components. Regarding quality and availability of features for usability, the cross-platform frameworks are rated good throughout, supporting the basic features as listed before (e.g. transitions).

The way how layouting works in the five solutions varies greatly, but with all of them, a customizable frontend can be created. Titanium additionally requires testing on both supported platforms because missing layouting attributes and different default behaviors can cause problems. The other two cross-platform solutions make porting very easy by using web technology. The subjective impression varies and the look of UI components differs greatly, which can be an important point when deciding between the solutions.

4.3 Developer support

Except for PhoneGap, which is used with separate native project files per platform and thus can be used with the respective IDEs, all other frameworks provide an IDE for development, debugging and deployment of applications. Rhodes is the only framework with a score of only 3 points because the learning effort for two programming languages (Ruby and JavaScript) is considered higher and Ruby code debugging is only supported on *RhoSimulator*, not on devices or other simulators. Typical requirements, such as debugging with breakpoints, are fulfilled by both the cross-platform frameworks and the native SDKs, rendering them pretty much equivalent for this category. Documentation quality and completeness is also adequate for serious development – solutions for Android and iOS SDK problems can be found more easily because of the large developer community.

4.4 Reliability & Performance

During implementation and testing of the sample application, no relevant crashes were found with any of the cross-platform frameworks and native SDKs, apart from memory management of loaded thumbnails from large pictures. With Titanium and the Android/iOS SDKs, references are managed manually, which led to crashes on the Android device because of the limited application memory. The native Android SDK solves this with functions to efficiently handle thumbnail loading which are however not available in Titanium. In these cases, the frameworks are still missing functionality. With the web technology based frameworks, memory management is performed automatically by the web view component and thus did not lead to similar issues.

Performance-wise, the cross-platform frameworks vary greatly from its native competitor. Both in observed reaction times and subjective regard, Titanium was the fastest of the

three cross-platform solutions, which, together with the native UI components, brings it close to the native SDKs. The choice of JavaScript, however, has certain implications: multithreading and asynchronous operations are not supported directly, apart from those APIs which are already implemented to run in the background (e.g. HTTP requests). PhoneGap as framework for use of web technology has this problem, while Rhodes and Sencha Touch have additional causes for their very poor performance[14]. PhoneGap's performance can be fast enough for many types of applications if the right web framework is chosen. Combined with Sencha Touch, it was very slow on the test devices, with screen switch times of up to multiple seconds. Rhodes has equally poor performance.

4.5 Deployment, Supportability, Costs

Each of the vendors of the tested cross-platform frameworks offers professional support with different options or payment plans. Google and the Open Handset Alliance do not seem to offer such plans for Android, and Apple only seems to offer e-mail support through a ticket system. Free support options such as online forums exist for all solutions, and a point that speaks for Android and iOS is that their developer community is so large that solutions to typical problems can almost always be found on the Internet and not only in the official forums. Even though support options differ, all five solutions can be considered very good in this category. One point was deducted from the iOS SDK score because of its only basic translation support[15] and the need to pay the membership fee to gain access to technical support and the bug tracker. Regarding internationalization capabilities, all frameworks are extensible so that external solutions can be used for internationalization and other purposes. Android already offers advanced support for translations.

5 Conclusion

A major advantage of the cross-platform frameworks is code reuse. In the evaluation, the sample application was always first implemented and tested on Android and only later we checked whether it runs unchanged on the iOS platform. Both HTML-based solutions, Rhodes and PhoneGap with Sencha Touch, did not require any changes to the code itself. Titanium, on the other hand, required small changes to the layouting attributes.

Regarding developer support, documentation and additional learning sources (screencasts, tutorials) are available for all evaluated solutions, and mostly of good quality even for the fairly young cross-platform frameworks. As the frameworks are under very active development, the quality can be expected to improve over time.

HTML-based framework types have specific advantages. For instance, PhoneGap requires close to no extra knowledge for an experienced web developer: project setup is simple

[14] Possible reasons are a complex and deeply nested HTML DOM tree with Sencha Touch and the client-server architecture with embedded web server of the Rhodes framework.

[15] Translation support was improved with the iOS SDK 6 which was not tested in this evaluation.

and applications can be developed with web technology only (unless PhoneGap plugins or platform-specific functionality are needed). In addition, an application can be deployed as normal web site if it can run without device features that are not available in a browser. Hence, PhoneGap is a good option if an application should also be the base for a mobile web site. Rhodes does not support web deployment in the evaluated version.

The main disadvantage of the cross-platform approach with the evaluated frameworks is the availability of usability features and the partially very poor performance as compared to the native implementations of our sample application. Rhodes and Sencha Touch do not support the native platform design which might be desired to present users with a more familiar interface, or to make sure an application is not rejected on app stores for disregarding UI guidelines. PhoneGap uses a better approach by allowing plugins to run native code and thus add platform-specific elements like the iOS tab bar. Regarding performance, it is notable that Facebook switched from mostly HTML5-based mobile applications (for Android and iOS) to native implementations and thereby achieved large improvements in UI responsiveness and overall performance [Dan12] [Du12].

In general, cross-platform solutions offer many advantages, in particular code reuse and quick setup for multiple platforms. If the mentioned usability and performance problems can be considered minor for an application, we can recommend to employ *Titanium* or *PhoneGap* for production use. Both frameworks can offer high performance[16]. Depending on the use case, considerations of desired user interface design (customized vs. native) and deployment as mobile web site, one framework has advantages over the other as described above. However, native development is still the better choice for usability-centered applications with a high focus on user experience. Also performance-critical applications, e.g. software performing heavy background processing, should be implemented natively. Nevertheless with modern devices, performance issues become less prevalent.

We expect the field of cross-platform mobile application frameworks to become appealing to developers through their ongoing development. As only little research was conducted in this area, many open questions exist: Will HTML5 become widely used when standards for accessing device features are implemented? How much quantitative influence on development/maintenance time and cost does the cross-platform approach have? Are other types of frameworks more suited to port an application to another platform? For instance, *XMLVM* provides translation of source code to a different target, and various other concepts exist. Also, our evaluation criteria were mostly based on aspects important for developers. Other viewpoints could be investigated as well. We could only test a small set of available solutions, and more frameworks may become ready for production use.

References

[A+09] Rama Akkiraju et al. Toward the Development of Cross-Platform Business Applications via Model-Driven Transformations. In *SERVICES I*, pages 585–592. IEEE Computer Society, 2009.

[16] PhoneGap's performance depends greatly on the HTML markup and related resources like CSS rules and JavaScript code. A fast web application framework, if at all required, is recommended for use with PhoneGap.

[A+10] Sarah Allen et al. *Pro Smartphone Cross-Platform Development: iPhone, BlackBerry, Windows Mobile, and Android Development and Distribution*. Apress, September 2010.

[Ado12] Adobe Systems Inc. PhoneGap. http://phonegap.com/, December 2012.

[App12] Appcelerator Inc. Appcelerator Titanium. https://www.appcelerator.com/platform, December 2012.

[BD09] Bernd Brügge and Allen H. Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java (Third Edition)*. Prentice Hall International, 2009.

[BH06] Judith Bishop and Nigel Horspool. Cross-Platform Development: Software that Lasts. In *Software Engineering Workshop, 30th Annual IEEE/NASA*, pages 119–122, April 2006.

[Dan12] Jonathan Dann. Under the hood: Rebuilding Facebook for iOS. https://www.facebook.com/notes/facebook-engineering/under-the-hood-rebuilding-facebook-for-ios/10151036091753920, August 2012.

[Du12] Frank Qixing Du. Under the Hood: Rebuilding Facebook for Android. https://www.facebook.com/notes/facebook-engineering/under-the-hood-rebuilding-facebook-for-android/10151189598933920, December 2012.

[HHM12] Frederick Hirsch and Dominique Hazaël-Massieux. Device APIs Working Group. http://www.w3.org/2009/dap/, December 2012.

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12*, 1990.

[Int12] International Data Corporation (IDC). Android- and iOS-Powered Smartphones Expand Their Share of the Market in the First Quarter, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS23503312, May 2012.

[jF12] The jQuery Foundation. jQuery Mobile. http://jquerymobile.com/, December 2012.

[MC08] Parastoo Mohagheghi and Reidar Conradi. An empirical investigation of software reuse benefits in a large telecom product. *ACM Trans. Softw. Eng. Methodol.*, 17(3), June 2008.

[Mot12] Motorola Solutions Inc. RhoMobile Suite. http://rhomobile.com, December 2012.

[Pas02] Alessandro Pasetti. *Software frameworks and embedded control systems*. Springer-Verlag, Berlin, Heidelberg, March 2002.

[Sen12] Sencha Inc. Sencha Touch. https://www.sencha.com/products/touch, December 2012.

[Syb11] Sybase Inc. Enterprise Mobility Guide 2011. http://www.sybase.de/mobilityguide, 2011.

[Vis12] VisionMobile. Developer Economics 2012. http://www.visionmobile.com/product/developer-economics-2012, June 2012.

[XML12] XMLVM. Overview: Android to iPhone. http://www.xmlvm.org/android, December 2012.

Debugging Cross-Platform Mobile Apps without Tool Break

Christoph Hausmann, Patrick Blitz, Uwe Baumgarten

c.hausmann@weptun.de, p.blitz@weptun.de, baumgaru@in.tum.de

Abstract: Besides its use in the web, the JavaScript programming language has become the basis of some of today's most important mobile cross-platform development tools. To enable and simplify debugging in such environments, this paper presents a novel method for debugging interpreted JavaScript code. The described method uses source code instrumentation to transform existing JavaScript programs in a way that makes them debuggable when executed in any standard JavaScript environment.

1 Introduction

In recent years, the number of applications that runs on smartphone operating systems like iOS or Android, commonly referred to as "apps", has grown tremendously[1]. However, developing such apps is quite complicated, as all common hardware platforms require a different toolset. To simplify this process, several cross-platform development tools have been developed recently. While often solving the issue of developing the business logic and a user interface for multiple platforms in an integrated fashion, no cross-platform development tool available at the time of writing provides a debugging approach which is directly integrated into the IDE. But for a high quality software product, it is important to be able to debug the created code [GHKW08]. Thus, we present a debugger which is integrated into a cross-platform development environment.

In section 2, we examine the existing work in the area of mobile cross-platform development tools. Based on this overview we derive the functional and non-functional requirements a debugger for such environments has to fulfill and compare them to the features of existing JavaScript debuggers. Section 3 describes our debugging approach, as well as the context of our work. The following section 4 shows the implementation of our concept using the existing cross-platform mobile development tool *AppConKit* as a basis. In section 5 we validate our implementation using an independent JavaScript test suite and discuss the results, before we end with a conclusion in section 6.

[1] http://www.mobilestatistics.com/mobile-statistics/

2 Existing Work and Requirements

Mobile Development Tools Examining the current landscape of software development tools for app development, we can separate them into five main groups as laid out in [WIM11]:

1. **Native development tools** from the operating system manufactures, e.g. the Android or iOS SDK. They all allow for on-device debugging [And] [iOS].
2. **Cross-compiler tools**. They might also allow for debugging, for example MonoTouch [Mon]. However, these cross-compiler tools do not allow for direct development of apps on multiple platforms - there is still effort involved in porting code between platforms.
3. **Web (HTML5) frameworks** like jQuery or Sencha Touch. These frameworks use a JavaScript debugger in the browser on the developer's PC to debug apps built with them. However, they do not provide an integrated debugger that allows for debugging the created apps directly on a mobile device. [jQu] [Sen]
4. **Hybrid app frameworks**. These combine native and web technology in one app. A common framework in that area, PhoneGap [Pho11], also does not provide means of debugging the apps on the mobile device.
5. **Application description languages** for cross-platform apps. An app created with the most prominent example of app description languages, the Appcelerator platform, cannot be debugged at all [App].

Requirements As long as software is written by humans, they will make mistakes leading to faulty or buggy programs [GHKW08]. These bugs are caused by errors in the source code, in the software design, and sometimes also by the compiler. The activity of analyzing those faults or bugs is called *debugging* [Ros96]. As outlined by Rosenberg ([Ros96]), context is very important for a debugger. To maximize the available context, the debugger should be integrated into the development environment, as all necessary information is available there. This has the added benefit of reducing the impact of tool changes .

The functional requirements for a debugger, according to the book *The Art of Debugging with GDB and DDD* by N. Matloff and P. Salzman, facilitate *The Principle of Confirmation* [MS08, p. 2]. This principle states, that a debugger helps confirming that all assumptions a programmer makes about his program are really true during runtime. To verify these assumptions, most debuggers provide the following set of features:

1. **control** the program flow.
2. **inspect** the contents of variables.
3. **modify** the contents of variables.

Our debugger will be implemented for the JavaScript programming language. JavaScript is a high-level, dynamic and untyped programming language with support for both object-oriented and functional programming styles. It is an interpreted language, which derives its syntax from Java. However, it is not related to Java in any way. [Fla11, p. 1] While JavaScript was made famous for its use as a scripting language for the web, it can be embedded into existing applications to add scripting support to them. On all current mobile operating systems, a powerful JavaScript engine can be run.

Our research shows that the non-functional requirements for an integrated mobile cross-platform debugger are:

- **Integrated User Interface**: The UI of the debugger should be integrated into an existing Integrated Development Environment (IDE) to minimize tool breaks and maximize the available context.
- **Low Response Time**: The UI of the debugger should feel responsive and the delay between firing a command and its response should not be higher than 500 ms on average, based on the Truthful Debugging principle stated in [Ros96].
- **Reliability and Integrity**: It should follow the basic principles of a debugger (Heisenberg Principle, Truthful Debugging, Context is the Torch in a Dark Cave) as described in *How Debuggers Work: Algorithms, Data Structures, and Architecture* [Ros96] .
- **Maintainability and Extensibility**: Its design and protocols should be well defined to allow future extensions.
- **Interoperability**: We can not make assumptions or changes in the runtime environment (JavaScript engine) that will run the code to support debugging. Hence debugging has to be supported with any JavaScript engine.

While the functional requirements are the basic requirements for every modern debugger for high-level languages, the non-functional requirements are specialized for the use case of debugging cross-platform applications.

Existing Debuggers There are several known solutions for debugging JavaScript like the Mozilla Rhino debugger [Rhi], Firebug [Fir] or the Chrome V8 debugger [Chr]. These solutions serve their purpose and have proven to work. However, there is one problem: most of them depend on a specific JavaScript engine. In fact, we found only one solution that provides the full set of debugging features but (at least in theory) does not depend on a specific engine: Crossfire [CB11]. But Crossfire requires that the JavaScript engine running the code implements the Crossfire commands and protocol, which violates the non-functional requirement of **Interoperability**. Hence the current situation is illustrated in figure 2.1. Every JavaScript engine we have looked at comes with its own debugger and defines its own remote debugging protocol.

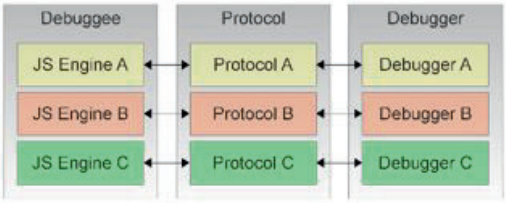


Figure 2.1: Architecture of existing remote debugging solutions for JavaScript.

3 Approach

Proposed Solution As no existing debugger fulfills our requirements, we propose an alternative solution to the challenge of implementing a platform-independent remote debugging solution for JavaScript. We call it the *Universal JavaScript Debugger.*

The basic idea behind the Universal JavaScript Debugger is the following: we know, that there is a JavaScript runtime environment on the debuggee-side[2]. So, if the debuggee-side of the debugging solution itself is implemented using JavaScript, the debugger supports any standard JavaScript engine. This way, we achieve three goals:

1. The solution works with any standard JavaScript engine.
2. There is no need to make changes to an existing JavaScript engine.
3. It can be integrated into an existing IDE.

To achieve this, we modify (instrument) the JavaScript source code before it is sent to the debuggee. As shown in figure 3.1, using that approach, there is only one debugger and only one protocol needed to provide debugging capabilities for various existing JavaScript engines.

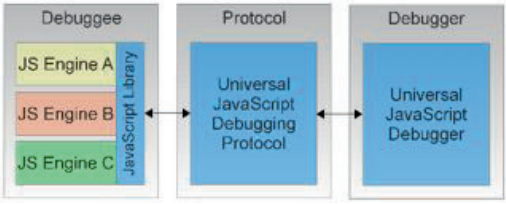


Figure 3.1: Architecture of the Universal JavaScript Debugger.

[2]The program which is being debugged by a debugger is referred to as the debuggee.

Our solution consists of the following parts:

- A **set of source code modification rules** on how to modify an existing JavaScript program to make it debuggable, i.e. to allow for attaching a debugger to the running program and analyzing its state and behavior.
- An **implementation** of those rules using a JavaScript parser and code generator.
- A **JavaScript library** for controlling the program flow and communicating with a remote debug UI.
- A **debug UI** based on the Eclipse software development platform for controlling the debuggee which will be integrated into the *AppConDeveloper* IDE.

Context The concept and implementation of our debugger is based on the AppConKit, a commercial cross-platform native app development tool created by Weptun[3]. We extended it to allow for debugging directly on the mobile device.

The AppConKit is a framework dedicated to the design and implementation of native mobile business applications for touchphones and tablets. It currently supports the iOS operating system from Apple (iPhone and iPad), as well as the Android platform developed by the Open Handset Alliance, where Google is one of the main contributors.

Supported by a graphical interface, mobile application developers are able to create native multi-platform mobile applications using a set of ready-to-use software components provided by the AppConKit. The AppConKit is based on a client-server architecture consisting of the *AppConClient* and the *AppConDeveloper* (or the *AppConServer* after deployment). During development of a mobile application, the AppConClient runs on the mobile device, while the AppConDeveloper runs on the developer's machine.

The challenge of developing applications for heterogeneous platforms and devices is solved by using an abstract platform-independent application description language. This application description, which is directly and automatically generated from the graphical representation of the application, is then interpreted by a special runtime on the mobile device.

In the most recent iteration, the AppConKit has been extended by adding the possibility to express the app's business logic using JavaScript code which is directly executed on the device. The architecture of the AppConKit can be seen in figure 3.2.

[3] `http://www.weptun.de`

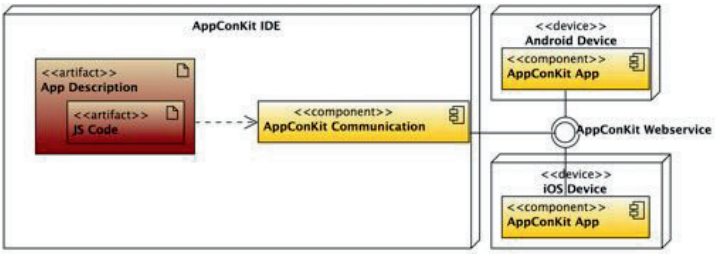


Figure 3.2: Architecture of the AppConKit.

4 Implementation

To evaluate the proposed solution, we implemented a prototypical system and tested it to show that we meet the goals outlined above.

System Architecture The abstract system architecture of the Universal JavaScript Debugger is shown in figure 4.1. On the debuggee-side there is a JavaScript engine which executes the previously instrumented JavaScript code. This instrumented code communicates with a JavaScript debug library, that also runs on the debuggee-side. This library exchanges information with a remote JavaScript debug connector located on the debugger-side. The connector is attached to a controller which is operated by the UI. On the debugger-side there is also the original unmodified JavaScript code written by the developer. It is needed to visualize the current state of the program, i.e. the lines where breakpoints are defined, and the current position where the program is suspended.

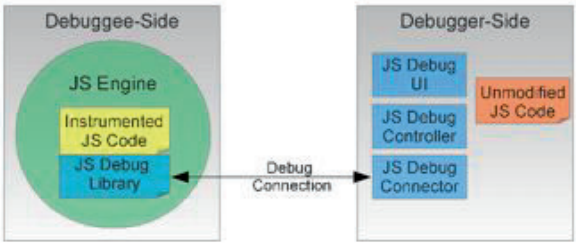


Figure 4.1: Abstract system architecture of the Universal JavaScript Debugger.

The actual system architecture can be seen in figure 4.2. The integration of the Universal JavaScript Debugger into the AppConDeveloper is carried out by extending the Eclipse JSDT [Ecl] environment.

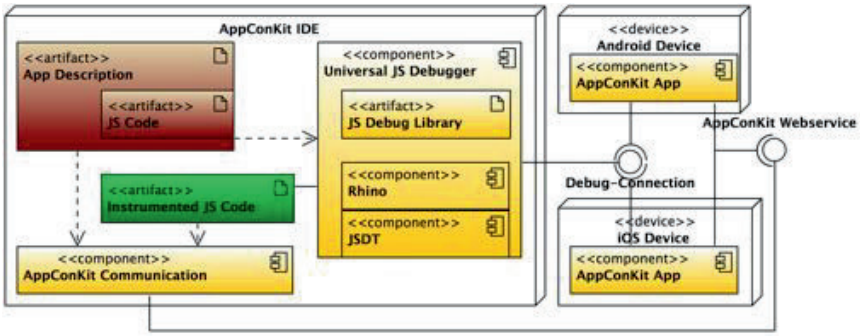


Figure 4.2: System architecture of the AppConKit including the integrated JavaScript debugger.

Integration into the AppConKit The AppConKit development environment can be used in either *Debug* or *Normal* mode. Our debugging procedure is only activated in debug mode.

When using the AppConKit in debug mode, the resulting design as shown in figure 4.2 is basically achieved by combining the designs of figure 3.2 and figure 4.1. We start with the AppConDeveloper, the Eclipse-based development environment used to implement the mobile applications. With the AppConDeveloper, the business logic can be implemented using JavaScript and attached to the mobile application, so that it can be executed on the AppConClient. However, before the app description is sent to the client, the JavaScript code is processed and instrumented as shown below. After that it is embedded into the app description, along with a JavaScript debug library. Upon interpretation of the app description on the debuggee-side, the embedded JavaScript code is extracted and executed within a JavaScript engine. During execution, the JavaScript debug library communicates with the Universal JavaScript Debugger within the remote AppConDeveloper via the debug connection. The AppConDeveloper now has the role of the debugger.

The actual source code modifications were implemented with a JavaScript parser, which creates an Abstract Syntax Tree (AST) of the program. In the next step, the AST is traversed, and calls into a debug library are inserted. Additionally, in order to allow for more sophisticated debugging features, certain patterns in the AST are identified and transformed according to a set of rules. In the final step, the modified AST is transformed into instrumented JavaScript source code by a code generator. We use the Mozilla Rhino environment [Moz] as the JavaScript parser and source code generator.

Instrumentation and Code Rewriting Rules We stated that we use source code instrumentation and other modifications to add additional statements to the source code. In this section, we will highlight the rewriting rules used to modify the code for instrumentation. In the following, we will refer to the original non-modified script as *original script*, and to the modified script as *instrumented script*.

Using source code instrumentation and other modifications of the original script, we want to achieve two goals:

1. The instrumented script gathers additional information about its state at runtime. This information then can be transferred to a remote debugger.
2. The instrumented script can be controlled by a remote debugger. This enables the implementation of breakpoints and step-by-step execution.

In order to reach these goals, we must be able to monitor and alter the behavior of the original script at a very fine-grained level. So the purpose of the following source code modifications is to be able to insert additional function calls at arbitrary locations in the original script. These additional function calls invoke functions of our JavaScript debug library, which are used to gather runtime information, and also to change the original script's behavior. To achieve this, we need to break up complex constructs in the original script into a set of more or less atomic operations. Between these atomic operations, additional statements have to be inserted, which capture the program state before and after those operations.

The two most important statements we insert are the inclusion of the debug library and the debug() statement itself. The debug library is used to manage the gathered runtime information and to communicate with the remote debugger. So the basic step of the source code modification consists of inserting the debug library at the beginning of the original script, which makes it globally available. The functions provided by the library are:

- **debug()** - used by the instrumented script in order to provide runtime information to the debug library. Additionally, the debug library is able to pause execution of the instrumented script by not returning from this function.
- **push()** - used to create a new stack frame and save the associated local variables whenever a new function is called.
- **pop()** - used to remove the top most stack frame and inserted whenever a return statement is found.

Now that the debug library is available, we need to insert a *debug()* call before every executed statement. Events can occur at any statement, and as such we must be able to pause execution at any statement. The formal definition of this transformation rule is very simple, it is shown in table 1.

Source Pattern	**Transformation Rule**
Statement	**debug(...);** *Statement*

Table 1: Source pattern and transformation rule for *Inserting Debug Statements* rule.

In total, we created 7 rules to cover all atomic operations listed in table 2. With the total source code instrumentation rule set we were able to cover every use case outlined there.

5 Validation

Debugger Tests We created test cases to verify that each of the debugger's functions works with every JavaScript source construct. Table 2 shows the test matrix, as well as the final test results. For each row in the test matrix, we implemented a JavaScript function, which uses the JavaScript source construct to be tested. After that, we executed this function six times, each time using a different debugger functionality (*Step In*, *Step Over*, *Modify Variable*, ...).

The test cases were executed using both iOS and Android as a debuggee and with the AppConDeveloper as the debugger. This ensures that no platform-specific errors exist, and that the functional requirements are fulfilled. Additionally, this way also the non-functional requirements **Integrated User Interface** and **Interoperability** can be validated.

			Debugger Functionality					
			Set Breakpoint	Step In	Step Over	Step Out	Inspect Variable	Modify Variable
JavaScript Source Construct	Choice:	If Else	✔	✔	✔	✔	✔	✔
		Switch Case	✔	✔	✔	✔	✔	✔
	Loops:	While	✔	✔	✔	✔	✔	✔
		Do While	✔	✔	✔	✔	✔	✔
		For	✔	✔	✔	✔	✔	✔
		For In	✔	✔	✔	✔	✔	✔
		For Each In	✔	✔	✔	✔	✔	✔
	Special Statements:	Break	✔	✔	✔	✔	✔	✔
		Continue	✔	✔	✔	✔	✔	✔
		Return	✔	✔	✔	✔	✔	✔
		Throw	✔	✔	✔	✔	✔	✔
		With	✔	✔	✔	✔	✔	✔
	Exceptions:	Try Catch	✔	✔	✔	✔	✔	✔
	Functions:	Definition	✔	✔	✔	✔	✔	✔
		Function Call	✔	✔	✔	✔	✔	✔
		Nested Function Calls	✔	✔	✔	✔	✔	✔
	Variables:	Declaration	✔	✔	✔	✔	✔	✔
		Assignment	✔	✔	✔	✔	✔	✔
	Data Types:	Number	✔	✔	✔	✔	✔	✔
		String	✔	✔	✔	✔	✔	✔
		Boolean	✔	✔	✔	✔	✔	✔
		Object	✔	✔	✔	✔	✔	✔
		Array	✔	✔	✔	✔	✔	✔
		Function	✔	✔	✔	✔	✔	✔
		null	✔	✔	✔	✔	✔	✔
		undefined	✔	✔	✔	✔	✔	✔

Table 2: Test matrix and results of debugger related test cases.

Mozilla JavaScript Test Library To verify that the behavior of the original scripts is not changed by our source code modifications and hence to validate the non-functional requirement **Reliability and Integrity**, we used the *Mozilla JavaScript Test Library*[4]. This test suite was created to test the functionality of the core JavaScript engine. It currently consists of 5216 test cases which verify the engine's behavior when executing a JavaScript program. The test cases itself are written in JavaScript, and they are executed using the Java Test Framework *JUnit*[5] and Mozilla Rhino as a JavaScript engine. There is a common JavaScript library used in every test case which provides the basic test infrastructure, e.g. means of comparing the expected and actual behavior, and raising an exception, if the test case has failed. The test cases are executed sequentially and upon completion, a test report is generated, which contains a list of all executed test cases with their status. There are three states:

1. **Passed** - expected and actual behavior are the same, everything works.
2. **Failed** - actual behavior differs from expected behavior, something is wrong.
3. **Error** - there was an error when executing the test case, e.g. a Java exception raised by the Mozilla Rhino JavaScript engine.

To be able to use this large test suite to verify the correctness of our code instrumentation rules, we modified it in the following way:

- When a JavaScript test case is read from file, we process the contents of the file first using our code instrumentation implementation, before handing it over to the test executor. This way, all executed code is instrumented JavaScript code.
- As instrumenting the code takes a few seconds, running the whole test suite sequentially can add up to a few hours in total. In order to verify the correctness of changes to the instrumentation code faster and more often, we further modified the test suite, so that several test cases are run in parallel. On a quad-core machine with 8 parallel threads (using hyper-threading) we could reduce the time for running the complete test suite to less than 45 minutes, which turned out to be an acceptable delay.

To make sure to only discover actual errors within the code instrumentation rules and implementation, we defined a three tiered test methodology, which allows us to ignore errors within third-party components, i.e. the JavaScript engine, or the JavaScript parser.

1. Run: execute the original test cases. This reveals all errors within the JavaScript engine itself. (Test cases: 5216, Passed: 96.3 %, Failed: 3.7 %, Error: 0 %)
2. Run: leave out the failed test cases of the first run. Before executing a test case, parse the JavaScript code, but don't modify the resulting AST. Instead, just generate JavaScript code again and run it. This reveals all errors within the parser and the code generator we use. (Remaining test cases: 5022, Passed: 90.6 %, Failed: 9.4 %, Error: 0 %)

[4] `http://www-archive.mozilla.org/js/tests/library.html`
[5] `http://www.junit.org/`

3. Run: leave out the failed test cases of the second run. For each remaining test case, instrument the original test case, and run the instrumented test case using the test executor. Now we know exactly which test cases failed due to the source code instrumentation, and not because of other errors. (Remaining test cases: 4548, Passed: 97.2 %, Failed: 1.7 %, Error: 1.1 %)

Test Results Figure 5.1 shows the results of the three runs in one chart.

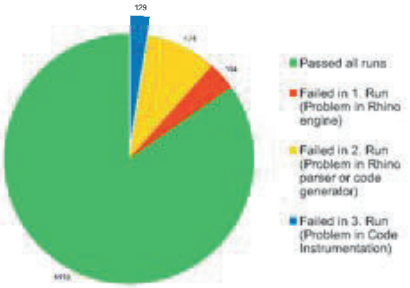


Figure 5.1: Test results achieved using the Mozilla JavaScript Test Library.

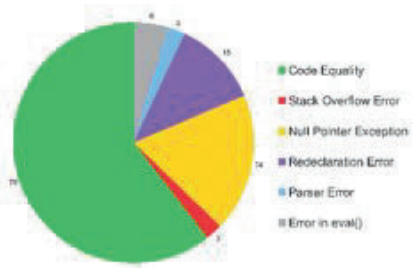


Figure 5.2: Detailed analysis of the test cases which failed due to code instrumentation.

Most of the errors are caused by either the Rhino engine itself, or Rhino's JavaScript parser, which we use. A further analysis of the failures revealed that most of the failures are caused by one of the following reasons:

- Missing dependencies: The test suite contains test cases designed to test functionality only available in a browser, such as means of modifying the HTML document currently displayed in the browser. As these features are not part of a standalone Rhino instance, all these test cases fail. This is the cause of most of the 194 failures by the Rhino engine.
- Code equality checks: In JavaScript, functions can be transformed into a String representation which contains the source code of that particular function. A lot of test cases make use of that feature to check whether the properties of that function match the expected content. These checks are very specific. In fact, it is enough to add an additional whitespace in order for such a test to fail. By using a parser and a code generator, the resulting source code is slightly different than the original one, which is the reason why using the Rhino parser alone is enough to make 474 tests fail. So these test cases can be considered invalid test cases, as they will also fail if there aren't any real problems in the source code.

With that knowledge in mind, we further analyzed the remaining 129 failures which were caused by the instrumented scripts. These are the failures we are interested in, because they

could be caused by incorrect code instrumentation rules. For our analysis, we assigned the failures to different categories, as figure 5.2 shows.

- 78 test cases have failed due to code equality checks. These test cases will never work, as by using source code instrumentation, the source code will always be different than what is expected by the test case.
- 27 test cases have failed due to errors within the JavaScript engine itself: 24 Null Pointer Exceptions and 3 Stack Overflow Errors within Rhino's Java code. Until now, we haven't identified the exact cause of these errors in Rhino's source code, however, they are likely to be fixed in future versions.
- 15 test cases failed due to a *Redeclaration Error* raised by the JavaScript engine. This turned out to be a bug in Rhino's implementation of the *eval()* function. We filed a bug report[6] for this error, and there is already a fix available.
- 6 test cases failed due to an incorrect behavior of the *eval()* function in some cases. There seems to be a problem with global objects defined within an *eval()* call. We are still investigating this problem, however, it seems to be an error within Rhino's *eval()* implementation, just like the *Redeclaration Error*.
- 3 test cases failed due to an incorrect handling of a reserved word by Rhino's JavaScript parser.

We were able to either solve every problem which resulted in a failed test case, or to trace back the problem to a third party component. Table 3 contains a summary of the final test results. It shows, that all remaining failing tests are caused by one of the following:

- Problems in the JavaScript engine.
- Problems in the JavaScript parser.
- The test procedure, e.g. by test cases which rely on completely unmodified source code.

Number of Test Cases	**Percentage**	**Result**	**Reason**
4419	84.7 %	Passed	No problems
552	10.6 %	Failed	Invalid test cases: code equality checks
194	3.7 %	Failed	Invalid test cases: missing dependencies
51	1 %	Failed / Error	Problem in Rhino JavaScript engine

Table 3: Summary of the test results achieved using the Mozilla JavaScript Test Library and the Mozilla Rhino JavaScript engine.

[6] `https://bugzilla.mozilla.org/show_bug.cgi?id=784358`

With these results, we can show that our source code modification rules are correct in the sense that they preserve the functionality of the instrumented scripts. And given the sheer number of successful tests, our implemented solution performs well enough to be used in nearly all practical settings. This means that we have reached the non-functional requirement **Reliability and Integrity**.

6 Conclusion

In this paper, we presented a method for device-independent integrated debugging of JavaScript programs which is based on a mobile cross-platform app development framework. We elicited the important functional and non-functional requirements our solution has to fulfill. Based on these requirements, we designed a universal JavaScript debugger and integrated a prototypical implementation into the AppConDeveloper IDE. Using this solution, we have run extended functional tests to ensure that we meet all requirements. We also were able to show that our approach could be integrated into an existing IDE and be used to debug code running on a remote device in the same context as writing or running it. With this prototypical solution, we have proven that the approach is generally feasible.

However, our work is not finished yet. We have started to evaluate the system in real-life scenarios and plan to release it to a wider public in the near future. In this context, several other studies will be carried out to lead to the final goal of simplifying the development of integrated mobile apps in the same way that IDEs [KKNS84] did for normal application development.

References

[And] Debugging | Android Developers. `http://developer.android.com/tools/debugging/index.html`. [Online; accessed 2013-02-07].

[App] Appcelerator: JavaScript Debugging. `http://developer.appcelerator.com/question/27731/javascript-debugging`. [Online; accessed 2013-02-07].

[CB11] Michael G Collins and John J Barton. Crossfire: multiprocess, cross-browser, open-web debugging protocol. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 115–124, New York, NY, USA, 2011. ACM.

[Chr] Chrome Dev Tools Debugger Features. `http://code.google.com/p/chromedevtools/wiki/EclipseDebuggerFeatures`. [Online; accessed 2012-10-14].

[Ecl] JavaScript Development Tools (JSDT). `http://www.eclipse.org/webtools/jsdt/`. [Online; accessed 2012-08-29].

[Fir] JavaScript Debugger and Profiler : Firebug. https://getfirebug.com/javascript. [Online; accessed 2012-10-14].

[Fla11] David Flanagan. *JavaScript: The Definitive Guide: Activate Your Web Pages (Definitive Guides)*. O'Reilly Media, 2011.

[GHKW08] Thorsten Grötker, Ulrich Holtmann, Holger Keding, and Markus Wloka. *The Developer's Guide to Debugging (Google eBook)*. Springer, 2008.

[iOS] Xcode User Guide: Debug and Tune Your App. http://developer.apple.com/library/ios/#documentation/ToolsLanguages/Conceptual/Xcode_User_Guide/060-Debug_and_Tune_Your_App/debug_app.html. [Online; accessed 2013-02-07].

[jQu] How to Debug Your jQuery Code. http://msdn.microsoft.com/en-us/magazine/ee819093.aspx. [Online; accessed 2013-02-07].

[KKNS84] Benn R. Konsynski, Jeffrey E. Kottemann, Jay F. Nunamaker, and Jack W. Stott. PLEXSYS-84: An Integrated Development Environment for Information Systems. *Journal of Management Information Systems*, 1(3):64–104, 1984.

[Mon] Debugging MonoTouch. http://docs.xamarin.com/ios/Guides/Deployment%252c_Testing%252c_and_Metrics/Debugging_in_MonoTouch. [Online; accessed 2013-02-07].

[Moz] Rhino overview | Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Rhino/Overview. [Online; accessed 2012-08-29].

[MS08] Norman Matloff and P J Salzman. *The Art of Debugging with GDB and DDD*. No Starch Press, 2008.

[Pho11] Debugging PhoneGap. http://phonegap.com/2011/05/18/debugging-phonegap-javascript/, 05 2011. [Online; accessed 2013-01-03].

[Rhi] Rhino Debugger Features. https://developer.mozilla.org/en-US/docs/Rhino/Debugger. [Online; accessed 2012-10-14].

[Ros96] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. Wiley, 1996.

[Sen] How to Debug Sencha Touch 2 Apps. http://stackoverflow.com/questions/10127244/how-to-debug-sencha-touch-2-apps. [Online; accessed 2013-02-07].

[WIM11] Felix Willnecker, Damir Ismailovic, and Wolfgang Maison. Architekturen mobiler Multiplattform-Apps. In Stephan Verclas and Claudia Linnhoff-Popien, editors, *Smart Mobile Apps*, number 26 in Xpert.press. Springer DE, 2011.

Platform architecture portfolio
Comparison of 3 platforms (Android, iOS, mobile Ubuntu)

Marlo Häring

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg
marlohaering@googlemail.com

Abstract: In this paper I will introduce three different mobile operating systems and their related developing approaches. Each of these systems requires divergent technical infrastructures to develop mobile applications.

1 Introduction

Android and iOS are currently the leading operating systems because of their popularity. Figure 1 shows the dominance of these two systems. On the 3rd of January Mark Shuttleworth, who provides leadership for the Ubuntu operating system, announced a new smartphone interface for its popular operating system Ubuntu.

Although all operating systems have the same tasks, they differ in numerous areas and fulfill them with diverse approaches. This is particularly evident in the area of user experience. Mobile Ubuntu for example tries to transfer their popular unity environment besides desktop PCs and television also to smartphones. So it stands out from existing operating systems. In particular the developing progress of a mobile application requires different technical infrastructures and platforms.

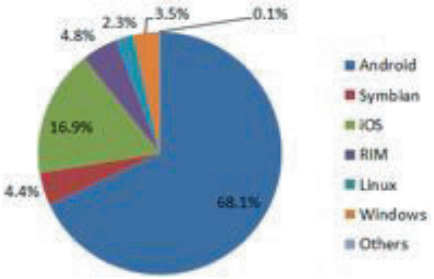


Figure 1: IDC looked at global smartphone shipments in Q2 2012, placing Android well out in front in terms of operating system shareSource: IDC Worldwide Mobile Phone Tracker, August 8, 2012

2 Developing for Android

Android apps can be developed on any common platform (Windows, Linux, MacOS) since *Eclipse* is the most popular IDE and it is mainly written in Java. Furthermore *Eclipse* brings a very mighty debugging engine as well as a plug-in infrastructure, which allows great flexibility. The Android Development Tools (ADT) are also installed as a plug-in. They include an emulator, which behaves like a real Android device and allows you to test your application without having a real device. It is even possible to create multiple Android Virtual Machine instances (AVM) with different customizations like display size, sd-card capacity or API Level. The emulator can run several AVMs in parallel. The whole backend code is written in Java. Android provides a subset of the Java 7 libraries adapted to the UI widgets and touch events that occur through the touch interface. The interface is defined in XML. The UI elements are structured hierarchically. *ViewGroups* are container. They define the layout structure of their containing *View* elements and can include further *ViewGroup* elements. In this way the GUI layout is designed. Eclipse brings a simple "*what you see is what you get*" GUI builder which parses and generates XML code. *DroidDraw*[1] is an alternative GUI builder also usable as an online applet version. To access UI elements in Java each element has a unique ID, which is defined in the XML structure. Subsequently action listener can be registered so that events are handled. When the app is built it is first compiled into *.class* files (Java-Byte code) and thereafter converted to *.dex* (Dalvik executable) files which can be executed by the Dalvik Virtual machine (VM). This VM creates an instance for each app running.

3 Developing for iOS

The developing of apps for Apple devices requires a Macintosh (MacOS). The developing environment consists basically of *Xcode*. It provides a very efficient simulator for IPhone and IPad as well as powerful debugging tools just like Microsoft's *Visual Studio*, *Eclipse* or Oracle's *Netbeans*. The sizes of apps developed are mostly very small so they can run on relative weak processors. For this reason the computer does not need to be powerful. It should have enough power for *Xcode* and especially for the simulator to run smoothly. Using the simulator the whole developing process can be accomplished exclusively on the mac so that no mobile devices are needed. The iOS version, display resolution and several other settings can be customized. User experience is the only field where the simulator is not suitable. Controlling the simulated device display by mouse doesn't feel as natural as by touch. The most inexpensive way to fill this gap is to buy an IPhone touch (roughly $300). Unfortunately a leak of sensors goes along with it but it is sufficient to feel the usage by finger swipes and accessibility of the UI widgets. The app is built with the *Model View Controller* (MVC) concept. The UI is designed by the *Xcode* integrated GUI builder and the backend parts (*Model* and *Controller*) are written in ObjectiveC. C and Smalltalk influenced this programming language so it brings object-oriented elements to C. Similar to C++ each class consists of

[1] http://www.droiddraw.org/

an interface file (.h) and an implementation file (.m). Properties are used to access the GUI within the ObjectiveC code. They can be created automatically with *Xcode* by dragging the respective element into the interface file. An outlet is created and can be used to interact with the UI element. Events are handled by delegate methods, which are methods that can be connected to a specific event. When the event occurs the connected method is invoked (delegate pattern).

4 Developing for mobile Ubuntu

Nowadays smartphones are produced, providing performance characteristics from former desktop PCs. This enables mobile operating systems to use modern multi core technology. As a result current graphic chips can render pictures for desktop PCs. With this technical development mobile Ubuntu forms in combination with a docking station, keyboard and monitor, a fully-fledged mobile workstation, which is expected to be very attractive to enterprise customers. Additionally mobile Ubuntu tries to attract smartphone beginners with a simple access to basic functionalities, used by the majority. Since mobile Ubuntu is a Linux system and has one of the best web service integration there are plenty of possibilities to develop an app. One option is the QML toolkit, which is built upon the newest version of the QT framework. The user interface consists of a tree of declared objects with properties. Similar to Android's UI elements QML objects have a unique ID, which is used to refer to. For dimensions QML uses grid units, which are device dependent. This is a helpful concept to build apps platform agnostic because layouts can be dimensioned once and used on multiple devices. A stricter JavaScript web browser variant can be embedded in several ways as a scripting language in QML. This framework provides a quick way to create interactive apps for Ubuntu on all devices.

5 Trends

Nowadays it becomes more and more important that web services are able to use system APIs to provide deep integration into the interface. The different kind of SDKs and frameworks make it difficult to develop cross-platform mobile apps, which appear same on different systems. As an intermediate step developer build their web services using HTML5, CSS and JavaScript and provide thin clients customized for each mobile system to access the programmed web service. By this way main parts of the app can be programmed generically as a website thus costs are saved because system independence is ensured. This development makes it obvious that web apps turn into first class citizens in mobile operating systems.

References

[And12] Android. Glossary. http://developer.android.com/guide/appendix/glossary.html, 2012. [Online; accessed 27-Jan-2013].

[Can12a] Canonical Ltd. Ubuntu app developer. http://developer.ubuntu.com/get-started/gomobile/, 2012. [Online; accessed 27-Jan-2013].

[Can12b] Canonical Ltd. Ubuntu for phones. http://www.ubuntu.com/devices/phone, 2012. [Online; accessed 27-Jan-2013].

[GR11] Mark H. Goadrich and Michael P. Rogers. Smart smartphone development - ios versus android.

[Qt12] Qt Project Hosting. JavaScript Expressions in QML Documents. http://qt-project.org/doc/qt-5.0/qtqml/qtqml-javascript-expressions.html, 2012. [Online; accessed 27-Jan-2013].

Develop and Scale Mobile Services in Cloud Computing Scenarios

Martin Ott

equinux
ott@equinux.com

Abstract: Mobile app development occurs in a highly competitive, and fast-moving environment. Small teams or even individual developers build apps that can be very successful, sometimes literally over night. Many of these successful apps are driven by backend services that are either interfaces to established web applications or that exist just for the purpose to serve the mobile apps. We discuss our experience with cloud computing scenarios which enable a small team to develop and scale mobile services for hundreds of thousands or even millions of users.

1 Introduction

Mobile services can be developed and deployed in various cloud computing scenarios. We distinguish the following scenarios as described in [VRMCL08] and [Fac13]: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Mobile Backend as a Service (MBaaS). In figure 1 we depict an exemplary 3-tier mobile web service architecture.

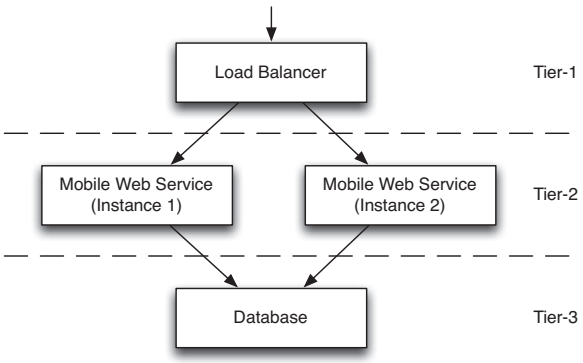


Figure 1: Example of a 3-tier mobile web service architecture

We use it as the common denominator in our discussion of cloud computing scenarios to compare how the tiers are implemented in the various scenarios.

Its public interface is based on the HTTP protocol. Tier-1 receives requests and dispatches them to instances of the mobile service on tier-2. This layer implements the public API of the service and runs one or more instances of the implementation. Tier-3 provides persistence for data of the mobile service by using a shared database.

This architecture has also been implemented in an IaaS-based, and a PaaS-based mobile service we have built, which serve as the backends for the iOS apps TV Movie [Mov13] and TV Movie HD [HD13]. At the time of this writing the mobile services powering these apps are in use by 875000 end-users.

On the basis of the exemplary architecture described above we discuss our experience with the programming interfaces offered in the different cloud computing scenarios, and how mobile services can be scaled by developers.

2 Programming Interfaces

Mobile apps are built on the APIs the vendors of mobile operating systems provide. When building mobile services in cloud computing scenarios, developers can also rely on APIs. For IaaS and PaaS scenarios their APIs provide ways to manage the infrastructure the service runs on. MBaaS service providers offer SDKs and APIs that are directly integrated into the mobile apps.

- `IaaS`: It offers compute and storage resources among other services as the basic building blocks. Vendors like Amazon Web Services [Ser13] let developers use SDKs and APIs to programmatically manage these resources. To build the described exemplary mobile web service architecture 4 compute instances with attached storage are required. On top of these compute instances developers would need to define and deploy their own software stacks, deploy a database of their choice and implement the mobile web service on top of that stack. Configuration management tools like Puppet [Pup13], and Chef [Che13] can help in the process of setting up and operating such stacks. On the one hand IaaS APIs make programmatic management of infrastructure possible and therefore lower the barrier for small teams to build distributed architectures for mobile services. On the other hand developers choose their complete software stack which lets them tune it for the specific requirements and purpose of the mobile service.

- `PaaS`: Predefined framework-specific and programming language-specific software stacks are deployed by the PaaS platform. Custom software stacks can be developed following the specifications given by the PaaS. The functionality of tier-1 is provided by the PaaS platform. Developers implement the mobile web service on tier-2 using the given software stack. Deployment workflows for the services in development are predefined. Developers can concentrate on developing the business logic of their service. APIs are provided from the PaaS platform to manage and interact with the running processes. Databases like the one we plan for tier-3 are often provided directly by PaaS platforms. For example, Heroku [Her13] provides PostgreSQL in

a SaaS scenario. Other databases are offered by various vendors. PaaS platform usually do not have specific SDKs in place for mobile services.

- `MBaaS`: Providers like StackMob [Sta13], Parse [Par13], Kinvey [Kin13], and FatFractal [Fat13] implement all three tiers of the architecture we have outlined above. The middle layer usually includes APIs to authenticate and manage users. The core is a REST-based API built on the principles described in [Fie00]. This API basically enables developers to read, write, update, and query arbitrary data. It acts as an persistence interface to tier-3 databases. Usually additional services like location queries, push notifications to clients, and the ability to run custom business logic from the app developers are provided in MBaaS scenarios. The API endpoints are not only implemented using REST-based protocols but are also provided as native SDKs for popular mobile operating systems. Developers can employ the same skill set for developing the mobile app and for interacting with the mobile service. On the other hand customization of the mobile service is either very limited or not possible at all.

3 Service Scalability

The widespread availability of mobile app stores, its top charts, and featured apps sections expose apps to many new users. This can result in fast user growth, even over a very short period of time like a few hours or days. When affected apps are backed by mobile services they need to be able to scale with this fast growth of users. For example, the backend service for the TV Movie app had to be scaled out over and over to accommodate the growing number of users. Since most of the mobile services implement their interfaces using HTTP-based protocols they can make use of architectures, techniques, and tools that have proven successful for the operation of large websites or web apps assuming the mobile service itself can run concurrently. Cloud computing tools promise to address these scalability needs:

- `IaaS`: Platforms like AWS EC2 and its corresponding services allow developers to manage the scalability needs down to compute and storage resources. For example, a mobile service that requires an in-memory database on tier-3 that cannot be scaled horizontally needs larger compute instances in order to scale up. IaaS often provides a multitude of specifically tuned compute resources for the different needs that mobile service architectures require. On the other hand scaling resources can be costly for developers because they need to manage their software stack right above the compute resources that IaaS offers. For example, scaling the web process layer on tier-2 usually requires not just to launch another process but to launch a new compute instance and dependent resources in order to run the new process.
- `PaaS`: Platforms like Heroku [Her13] or Google App Engine [Eng13] operate a routing layer which maps to tier-1 in our mobile web service architecture. This frees developers from maintaining tier-1. Developers can concentrate on designing tier-2 processes so that the service can scale out by controlling the number of running

instances. Tier-3 services like shared databases usually can not be deployed in PaaS environments. Software as a Service (SaaS) tools can be choosen as a replacement.

- `MBaaS`: The provider of the MBaaS platform transparently scales the mobile web service as needed. Developers do not have access to control any of the tiers that the MBaaS platform runs the mobile web service on.

4 Conclusion

The discussed cloud computing scenarios to build mobile services provide everything from basic infrastructure building blocks to SDKs that are directly embedded into mobile apps. On one end of the spectrum mobile services can be built on top of IaaS platforms where they can be tailored exactly for the specific requirements developers have for their products. Scaling is performed directly at the infrastructure level resources which are composed by developers to the desired architecture. But scaling operations are usually supported by automated mechanisms. MBaaS offerings on the other end of the spectrum enable developers to start with predefined services that can be accessed with native SDKs. They are scaled transparently. PaaS platforms provide a middle ground where the infrastructure, deployment workflows, and scaling schemes are predefined and developers need to implement the API for their mobile services using 3rd-party or custom frameworks.

References

[Che13] Chef. `http://www.opscode.com/chef/`, 2013. [Online; accessed 15-January-2013].

[Eng13] Google App Engine. `https://developers.google.com/appengine/`, 2013. [Online; accessed 15-January-2013].

[Fac13] Michael Facemire. Mobile Backend as a Service. `http://blogs.forrester.com/michael_facemire/12-04-25-mobile_backend_as_a_service_the_new_lightweight_middleware`, 2013. [Online; accessed 05-February-2013].

[Fat13] FatFractal. `http://fatfractal.com`, 2013. [Online; accessed 15-January-2013].

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[HD13] TV Movie HD. `http://www.equinux.com/goto/tvmoviehd`, 2013. [Online; accessed 15-January-2013].

[Her13] Heroku. `http://www.heroku.com`, 2013. [Online; accessed 15-January-2013].

[Kin13] Kinvey. `http://www.kinvey.com`, 2013. [Online; accessed 15-January-2013].

[Mov13] TV Movie. http://www.equinux.com/goto/tvmovie, 2013. [Online; accessed 15-January-2013].

[Par13] Parse. https://www.parse.com, 2013. [Online; accessed 15-January-2013].

[Pup13] Puppet. http://puppetlabs.com/puppet/puppet-open-source/, 2013. [Online; accessed 15-January-2013].

[Ser13] Amazon Web Services. http://aws.amazon.com, 2013. [Online; accessed 15-January-2013].

[Sta13] StackMob. https://www.stackmob.com, 2013. [Online; accessed 15-January-2013].

[VRMCL08] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

Modellierung von Vorgehensmodellen – Paradigmen, Sprachen, Tools

Modellierung von Vorgehensmodellen – Paradigmen, Sprachen, Tools

Marco Kuhrmann[1], Daniel Méndez Fernández[1], Oliver Linssen[2], Alexander Knapp[3]

[1]Technische Universität München, Fakultät für Informatik, 85748 Garching
{kuhrmann,mendezfe}@in.tum.de

[2]Liantis GmbH & Co. KG, 47798 Krefeld
oliver.linssen@liantis.com

[3]Universität Augsburg, Institut für Informatik, 86159 Augsburg
knapp@informatik.uni-augsburg.de

Vorwort: Umfangreiche Vorgehensmodelle, wie sie in großen Organisationen etabliert sind, dienen als Quelle für Strukturen, Informationen und Wissen, welches in Software- und Systementwicklungsprojekten angewendet werden kann. Dazu müssen solche Vorgehensmodelle in der Lage sein, die in ihnen abgelegten Informationen für ihre Konsumenten (Projektmanager, Teammitglieder, Auditoren, etc.) einfach zugänglich zu machen. Modellierer (Prozessingenieure) müssen aber auch in der Lage sein, Informationen in einer strukturierten Weise zu erfassen. Auch müssen die Vorgehensmodelle der Tatsache Rechnung tragen, dass in einer Organisation ggf. viele Projekte durchgeführt werden, die sich in ihren Rahmenbedingungen teils grundlegend unterscheiden können. Ein Mechanismus zur Anpassung und zur projektspezifischen Instanziierung ist dafür essenziell. Darüber hinaus sollten die Vorgehensmodelle in einer Art repräsentiert werden, die es ermöglicht, sie einfach in Werkzeugen zu implementieren.

Vorgehensmodelle müssen somit vielen Anforderungen gerecht werden, die sich durch die „klassische“ Repräsentation als „Buch“ nicht erfüllen lassen. Etablierte Vorgehensmodelle wie der Rational Unified Process, Hermes oder das V-Modell XT sind daher als Modell konstruiert. Die Erfahrung zeigt, dass die Komplexität solcher Modelle ein Maß annimmt, das nur noch wenige Spezialisten vollumfänglich erfassen und das von Projektteams i.d.R. als Belastung empfunden wird. Das V-Modell XT z.B. besteht in der aktuellen Version 1.4 aus mehreren Tausend Modellelementen, die in einer fast 1.000-seitigen Dokumentation münden. Im Rahmen der Prozessanpassung bzw. der Weiterentwicklung und Pflege entstehen somit immense Aufwände allein in der Sicherstellung der Konsistenz.

Moderne Vorgehensmodelle zeigen, dass die Modellierung ein probates Mittel ist, um die Komplexität besser zu beherrschen. Gleichzeitig stoßen alle Vorgehensmodelle immer wieder an ihre Grenzen, wenn es z.B. um die Ausführung von Vorgehensmodellen (Enactment), die dynamische Anpassung zur Projektlaufzeit (Tailoring) oder die Auditierung von Ist-Prozessen gegenüber einem Referenzmodell geht.

Inhalte des Workshops

In den für den Workshop ausgewählten Beiträgen werden unterschiedliche Fragestellungen zur Beschreibung und Modellierung von Vorgehensmodellen besprochen. Einen Schwerpunkt bilden hierbei das Method Engineering, das Enactment und weitere Themen zu:

- Paradigmen zur Modellierung von Vorgehensmodellen
- Modellierung von reichhaltigen und agilen Vorgehensmodellen
- Repräsentation und Visualisierung von Vorgehensmodellen
- Artefaktorientierung als Konzept zum Aufbau von Vorgehensmodellen
- Method Engineering als Konzept zum Aufbau von Vorgehensmodellen
- Wandlungsfähige Vorgehensmodelle

Die ausgewählten Beiträge des Workshops sind ein Anstoß zur Diskussion zu aktuellen Entwicklungen zur Modellierung von Vorgehensmodellen. Insbesondere werden neben dem „State-of-the-Art" Forschungsaktivitäten und Konzepte, bzw. Prototypen für die Unterstützung von Vorgehensmodellen zur Projektlaufzeit besprochen. Ziel dieses Workshops ist es, den aktuellen Stand in der Modellierung von Vorgehensmodellen festzustellen und Potenzial für die weitere Forschung zu identifizieren.

Besonderen Dank wollen wir an dieser Stelle den Freiwilligen des Programmkomitees aussprechen, die maßgeblich zur Auswahl inhaltlich hochwertiger Beiträge beigetragen und uns die Auswahl ein wenig erleichtert haben. Vielen Dank!

Dem Programmkomitee gehörten folgende Personen an:

Jens Calamé	SQS AG, Köln
Dr. Gerhard Chroust	J. Kepler University Linz
Masud Fazal-Baqaie	Universität Paderborn
Ulrike Hammerschall	FH München
Eckhart Hanser	DHBW Lörrach
Patrick Keil	TU München
Alexander Knapp	Universität Augsburg
Ralf Kneuper	freiberuflicher Berater, Darmstadt
Marco Kuhrmann	TU München
Oliver Linssen	Liantis GmbH & Co. KG
Martin Mikusz	Universität Stuttgart
Daniel Méndez Fernández	TU München
Jürgen Münch	Univerity of Helsinki
Doris Rauh	Siemens AG, München
Andreas Rausch	TU Clausthal
André Schnackenburg	Bundesverwaltungsamt, BVA/BIT, Köln
Doris Weßels	FH Kiel

Wir bedanken uns bei allen Beteiligten des Workshops und der Tagungsleitung vor Ort in Aachen.

Marco Kuhrmann, Daniel Méndez Fernández,
Oliver Linssen und Alexander Knapp

Modellierung und Enactment mit ESSENCE

Michael Striewe, Michael Goedicke
Paluno - The Ruhr Institute for Software Technology, Universität Duisburg-Essen
Gerlingstraße 16, 45127 Essen
{michael.striewe,michael.goedicke}@s3.uni-due.de

Abstract: Komplexe Softwareentwicklungsmethoden entstehen selten in einem Zug und sind danach unveränderlich, sondern werden schrittweise erstellt und kontinuierlich verbessert. Der konkrete Projektfortschritt muss durch sie messbar sein, aber gleichzeitig muss das Vorgehen projektspezifisch adaptierbar bleiben. Vor dem Hintergrund dieser Anforderungen wurde von der SEMAT-Initiative ESSENCE entwickelt, das den agilen Umgang mit Softwareentwicklungsmethoden besser ermöglichen soll als mit den bisher verfügbaren Ansätzen. Dieser Beitrag stellt die Kernkonzepte von ESSENCE vor.

1 Einleitung

Moderne Softwareentwicklung ist ohne komplexe Softwareentwicklungsmethoden kaum denkbar, da bei der Entstehung eines großen Softwareproduktes zahlreiche Schritte ausgeführt und zahlreiche Akteure koordiniert werden müssen. Die Modellierung und Dokumentation der gewählte Methode spielt dabei eine wichtige Rolle: Nur so kann sichergestellt werden, dass allen Beteiligten die notwendigen Schritte überhaupt bekannt sind, der Projektfortschritt kann gegen dokumentierte Zwischenziele gemessen werden und kritische Planabweichungen sowie Verbesserungspotenziale können schneller erkannt werden, wenn ein Vergleich zwischen Soll und Ist möglich ist. Bei komplexen Methoden wird eine solche Dokumentation aber auch selber komplex und damit schwer zu lesen und schwer wartbar; im Extremfall nur durch wenige Mitarbeiter in der Rolle eines Method Engineer. Dies verhindert gegebenenfalls eine zeitnahe und flexible Anpassung der Methoden, um auf individuelle Projektsituationen einzugehen, und schränkt damit den Nutzen von Methoden ein.

Um diesem Problem zu begegnen wurde auf Basis mehrjähriger praktischer Erfahrungen von der SEMAT-Initiative[1] ESSENCE entwickelt. ESSENCE stellt nicht nur Möglichkeiten zur Beschreibung von Softwareentwicklungsmethoden bereit, sondern auch zur Ausführung dieser Methoden, zur Adaption in laufenden Projekten und zur Beurteilung des Projektfortschritts. Philosophie, Bausteine und Sprachdesign von ESSENCE werden im Folgenden aufgezeigt und diskutiert. Dieser Artikel basiert dabei auf der jüngsten Version der ESSENCE-Spezifikation, die unter der Dokument-Nummer *ad/2012-11-01* bei der OMG zur Annahme als Standard eingereicht wurde.

[1] `http://www.semat.org/`

2 Das Konzept von ESSENCE

Um dem Problem der Komplexität aktueller Modellierungsansätze zu begegnen, trennt ESSENCE zunächst zwischen zwei Metamodellen, die bisher üblicherweise gemeinsam betrachtet werden: Während Ansätze wie SPEM [SPE08] und ISO 24744 [ISO07] ein sehr umfangreiches Metamodell anbieten, das sowohl die technische Infrastruktur der Sprache als auch die wichtigsten Modellierungskonzepte abbildet, trennt ESSENCE zwischen einem übersichtlicheren Metamodell der technischen Infrastruktur in Form syntaktischer Element (die „ESSENCE-Sprache“), und einer separat beschriebenen Menge der wichtigsten Konzepte und Begriffe als Metamodell für Methoden (der „ESSENCE-Kernel“) [KMSG+12]. Der Kernel und die daraus gebauten Methoden werden dabei in der Syntax der Sprache ausgedrückt, sind damit also selber bereits Instanz eines Metamodells, auch wenn ihre Elemente zur Ausführung des Vorgehensmodells selber noch einmal instanziiert werden. Der Modellierungsansatz greift damit die Dualität von Klasse und Objekt auf [HSGP05], bleibt aber völlig im Rahmen des Ansatzes von MOF zur Metamodellierung [ESMB12]. Ziel dieses Designs ist es, sowohl die Erstellung als auch die Nutzung von Methoden zu erleichtern. Wie diese Trennung genau vollzogen wird, wird in den beiden folgenden Kapiteln erläutert.

Bei der Erstellung des Kernels ist es nicht das Ziel, eine möglichst umfangreiche Terminologie oder Ontologie der Softwareentwicklung zu erstellen, zumal es dafür bereits ältere Ansätze gibt [SWE]. Vielmehr folgt ESSENCE der Philosophie, dass Perfektion dann erreicht ist, wenn es nichts mehr wegzunehmen gibt. Je kleiner und universeller der Kernel ist, umso eher ist er tatsächlich die Essenz dessen, was allen Entwicklungsmethoden zugrunde liegt, während die Ausgestaltung dieser Grundlagen in der Hand der einzelnen Methoden liegt. Analog dazu wird in der Beschreibung einzelner Methoden davon ausgegangen, dass auch hier zunächst einmal nur die essentiellen Bestandteile dieser Methode festgelegt werden sollen, während detailliertere Ausgestaltungen als projekt- oder unternehmensspezifische Erweiterungen geschehen.

3 Die Bausteine von ESSENCE

In der natürlichen Sprache kann ein Text in Sätze zerlegt werden und diese wiederum in Worte. Jedes Wort gehört dabei einer bestimmten Wortart an und die korrekte Bildung von Sätzen unterliegt grammatikalischen Regeln. Wortarten und Regeln sind typischerweise weitgehend starr und verändern sich nur in sehr langen Zeitabstände, während eine lebendige Sprache ständig neue Worte bilden und aus bestehenden Worten immer neue Sätze formen wird. Gleichzeitig gibt es in jeder Domäne wichtige Worte, die immer wieder verwendet werden. In Analogie zu dieser (zweifellos vereinfachten) Darstellung natürlicher Sprache führt die ESSENCE-Spezifikation mehrere Bausteine ein (siehe Abbildung 1): Das Metamodell der Sprache als Sammlung der Wortarten und grammatikalischen Regeln, den „Kernel“ als Sammlung wichtiger Konzepte, „Practices“ als abgeschlossene Vorgehensbeschreibungen für eine konkrete Problemstellung, und letztendlich Methoden als Komposition mehrerer Practices auf Basis eines gemeinsamen Kernels. In der MOF-

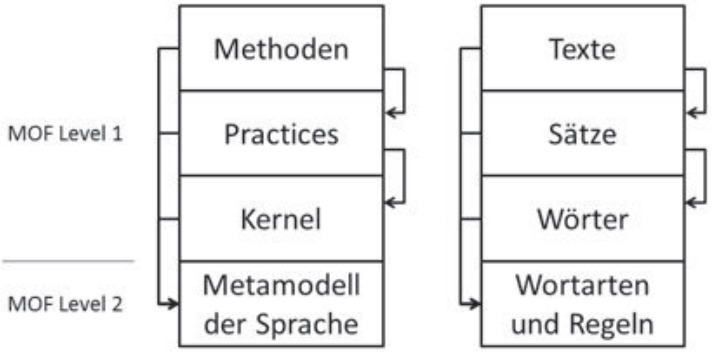


Abbildung 1: Die Bausteine von ESSENCE (links) in Analogie zu den Bausteinen natürlicher Sprache (rechts).

Hierarchie ist die Sprache auf Level 2 anzusiedeln, während sich Kernel, Practices und Methoden auf Level 1 befinden.

3.1 Das Metamodell

Abbildung 2 zeigt eine konzeptionelle Sicht auf das Metamodell der ESSENCE-Sprache und die wichtigsten darin deklarierten Typen. Bei den Elementen in der Mitte der Abbildung (*Alphas* und *Alpha States*, *Activity Spaces* und *Competencies*) handelt es sich um Elemente für die Beschreibung abstrakter, methodenunabhängiger Konzepte. Durch Verbindung mit den auf der rechten Seite aufgeführten konkreten Elementen (*Work Products* und *Activities*) können diese verschiedene Ausprägungen erhalten. Die außen dargestellten so genannten *Patterns* wiederum dienen als generisches Mittel zur Herstellung von beliebigen Bezügen zwischen abstrakten und konkreten Elementen.

Die Elemente *Alpha*, *Alpha State* und *Work Product* befassen sich dabei mit den Dingen, die Gegenstand eines Projektes sind. Beispielsweise kann das Alpha „Requirements“ die Anforderungen abstrakt repräsentieren und vom Zustand „Conceived“ bis zum Zustand „Fulfilled“ geführt werden, während die konkrete Repräsentation der Anforderungen in einem Projekt durch User Stories erfolgt, die demnach ein konkretes Work Product darstellen. Andere Projekte können sich für andere Work Products entscheiden, das Alpha bleibt jedoch dasselbe. Die Zustände eines Alphas sind dabei eines der entscheidenden Konzepte in ESSENCE: Jeder Zustand ist mit einer Anzahl von Checkpoints versehen, die jederzeit überprüft werden können. Auf diese Weise kann unabhängig von der verwendeten Methode ein Ziel formuliert und seine Erreichung überprüft werden. Im Laufe eines Projektes können sich alle Alphas unabhängig voneinander in ihren Zuständen vorwärts oder auch zurück bewegen. Die Namen der Alpha States sind ausnahmslos positiv formuliert, da es sich um gewünschte Zustände handelt, deren Erreichung für das Projekt wertvoll ist. In der Realität werden zweifellos auch ungünstige Zustände existieren, deren

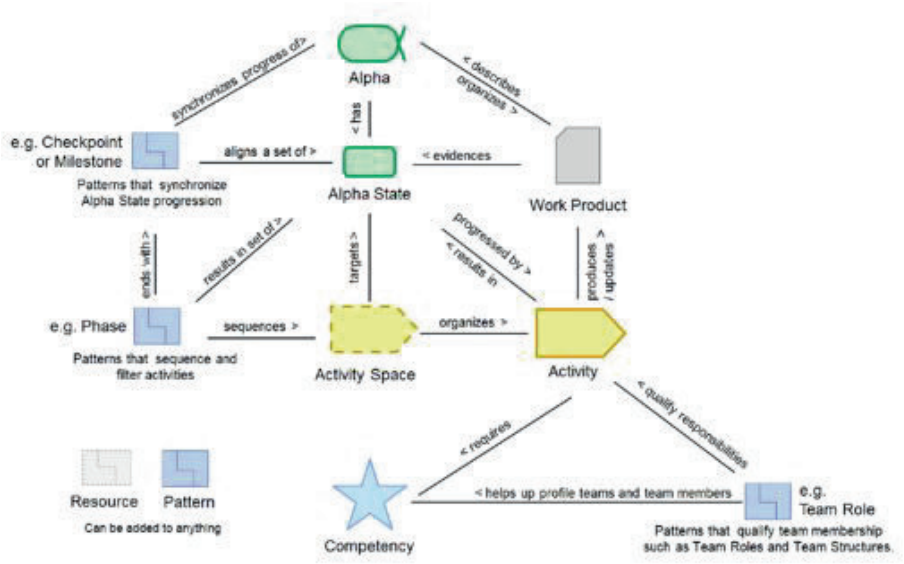


Abbildung 2: Konzeptionelle Sicht auf die zentralen Elemente des Metamodells der ESSENCE-Sprache (aus *ad/2012-11-01*). Nicht abgebildet sind Elemente für Kernel, Practices und Methoden, die im Wesentlichen Container für die abgebildeten Elemente sind.

explizite Benennung für die Erstellung von Methoden aber ohne Belang ist und die daher in ESSENCE nicht weiter betrachtet werden.

Die Elemente *Activity Space* und *Activity* befassen sich mit den Tätigkeiten, die in einem Projekt auszuführen sind. Auch hier gilt die Trennung zwischen abstrakter und konkreter Beschreibung, so dass z.B. der Activity Space „Implement the System“ abstrakt das gesamte Tätigkeitsfeld der Implementierung umfasst und mit konkreten Aktivitäten wie Codegenerierung, manueller Implementierung usw. gefüllt werden kann. Wissen, Fähigkeiten und Einstellungen, die zur erfolgreichen Durchführung der Tätigkeiten notwendig sind, werden wiederum abstrakt als *Competency* beschrieben.

Neben der direkten Zuweisung an Aktivitäten und Activity Spaces können diese insbesondere auch in *Patterns* verwendet werden, die zum Beispiel komplexe Rollen und Verantwortlichkeiten abbilden können. So kann beispielsweise modelliert werden, dass der Inhaber einer bestimmten Rolle über bestimmte Kompetenzen verfügen muss, für bestimmte Work Products verantwortlich ist, einen bestimmten Activity Space leitet und für die Erreichung eines bestimmten Alpha States verantwortlich zeichnet. Das Metamodell definiert dabei insbesondere keine grundsätzlichen Beschränkungen, wie detailliert und umfangreich derartige Beziehungen über Patterns gestaltet werden. Definiert ein Unternehmen seine Rollen also gänzlich anders, ist dies trotzdem mit demselben Mechanismus abbildbar. Genauso können beliebige andere Zusammenhänge über Patterns modelliert werden.

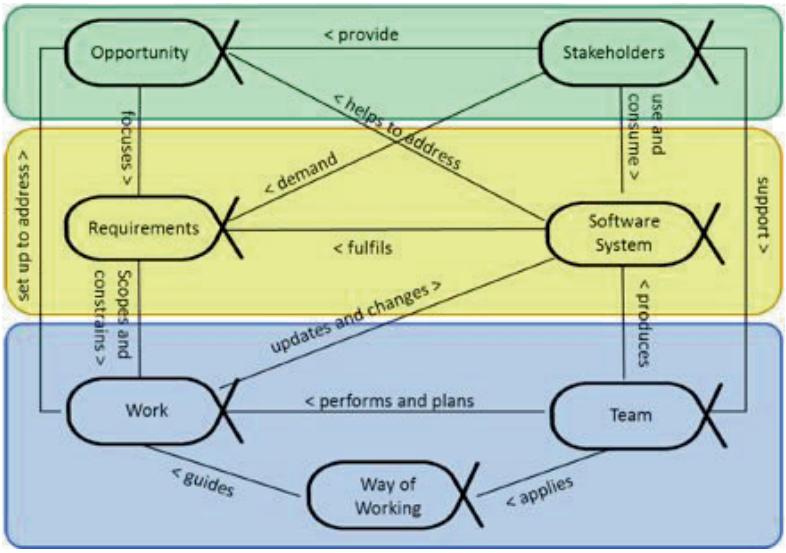


Abbildung 3: Die Alphas des Kernels und ihre Zusammenhänge (aus *ad/2012-11-01*).

3.2 Der Kernel

Der von der SEMAT-Initiative erarbeitete und zur Standardisierung in ESSENCE vorgeschlagene Kernel des Software Engineering enthält sieben Alphas, die die zentralen Aspekte des Software Engineering und ihre Zustände methodenunabhängig beschreiben: *Requirements*, *Software System*, *Team*, *Work*, *Way-of-Working*, *Stakeholder* und *Opportunity*. Diese Alphas und ihre Zusammenhänge sind in Abbildung 3 dargestellt. Die Alphas *Stakeholder* und *Opportunity* bilden dabei den unternehmerischen Bereich des Software Engineering ab, d.h. insbesondere die Schnittstelle zu Kunden, Nutzern und dem wirtschaftlichen oder sozialen Wert, der durch die Entwicklung eines Softwareproduktes geschaffen werden soll. Die Alphas *Requirements* und *Software System* bilden den technischen Aspekt der Softwareentwicklung ab, während sich *Team*, *Work* und *Way-of-Working* mit dem Management von Softwareentwicklung befassen und die Steuerung der Methode fokussieren. Dabei ist insbesondere zu berücksichtigen, dass über den Way-of-Working das modellierte Vorgehen selber mit in die Betrachtung eingeschlossen wird. Das Vorgehen, das die Anpassung des eigenen Vorgehens beschreibt, kann somit selber auch in ESSENCE ausgedrückt werden. Neben den Alphas enthält der von der SEMAT-Initiative vorgeschlagene Kernel 15 Activity Spaces zur Beschreibung zentraler Tätigkeitsbereiche, die sich ebenfalls auf die genannten drei Teilbereiche verteilen, sowie sechs Competencies.

Der Grundgedanke des Kernels bringt es mit sich, dass die genannten Elemente einigen Anwendern nicht ausreichend detailliert erscheinen oder ein für eine bestimmte Domäne wichtiges Element fehlt. Dies ist in der ESSENCE-Spezifikation berücksichtigt, indem der Kernel erweitert werden kann. Zusätzliche Alphas können entweder als eigenständi-

ge Elemente hinzugefügt werden, oder als untergeordnete Alphas ein vorhandenes Alpha weiter detaillieren. Die ESSENCE-Spezifikation schlägt mehrere solcher Erweiterungen für jeden der drei Teilbereiche vor. Beispiele dafür sind *Task* als weitere Detaillierung für *Work* oder *Component* als weitere Detaillierung für *Software System*. Dabei ist entscheidend, dass es sich bei diesen Erweiterungen nicht um eine Erweiterung des Metamodells der Sprache und damit einen Eingriff in die Sprachkonstruktion handelt, sondern dass die Kernel-Elemente und die Erweiterungen selber bereits Instanzen dieses Metamodells sind. Damit ist es grundsätzlich sogar möglich, mit dem Metamodell und der Ausführungssemantik von ESSENCE auch das Vorgehen in gänzlich anderen Domänen als der Softwareentwicklung zu modellieren, indem statt des oben beschriebenen Kernels andere Alphas identifiziert werden.

3.3 Die Practices

Als Zwischenschritt zwischen dem Kernel als Sammlung abstrakter Elemente und einer kompletten Softwareentwicklungsmethode verwendet ESSENCE das Konzept von Practices, die konkrete Handlungsempfehlungen für begrenzte Probleme des Software Engineering geben, z.B. die Verwendung von Use Cases zur Erhebung der Anforderungen. Sie ordnen den abstrakten Elementen des Kernels die jeweils angemessenen konkreten Ausprägungen zu und können bei Bedarf auch weitere Alphas einführen.

Wie detailliert eine einzelne Practice ausgestaltet wird, ist dem Autor der jeweiligen Beschreibung überlassen. Die Zuordnung eines einzelnen Work Products zu einem Alpha ist genauso eine valide minimale Practice wie die Zuweisung einer Aktivität zu einem Activity Space. Auch die Definition einiger Patterns, die Alpha States zu Meilensteinen gruppieren, ohne überhaupt Bezug auf konkrete Work Products und Aktivitäten zu nehmen, kann eine minimale Practice bilden. Lässt eine Practice Fragen unbeantwortet bzw. gibt sie für bestimmte Situationen keine Handlungsempfehlungen, kann diese Lücke entweder durch die Komposition mit einer anderen Practice oder durch spontane Adaption während der Durchführung des Projektes geschlossen werden.

3.4 Die Methoden

Auf Basis der gemeinsamen Elemente des Kernels können Practices miteinander kombiniert (oder als inkompatibel identifiziert) werden, um letztlich in der flexiblen und agilen Komposition und Konfiguration von Methoden zu münden. Dabei müssen gleichartige Elemente, die in verschiedenen Practices auftreten, miteinander verschmolzen werden, so dass eine Methode nicht einfach nur eine Sammlung von nebeneinander stehenden Practices ist.

Durch den Bezug zum Kernel kann schnell geprüft werden, ob eine passende Auswahl an Practices getroffen wurde, die in die Methode integriert werden. Zum Beispiel kann geprüft werden, ob allen Alphas mindestens ein Work Product zugewiesen wurde oder

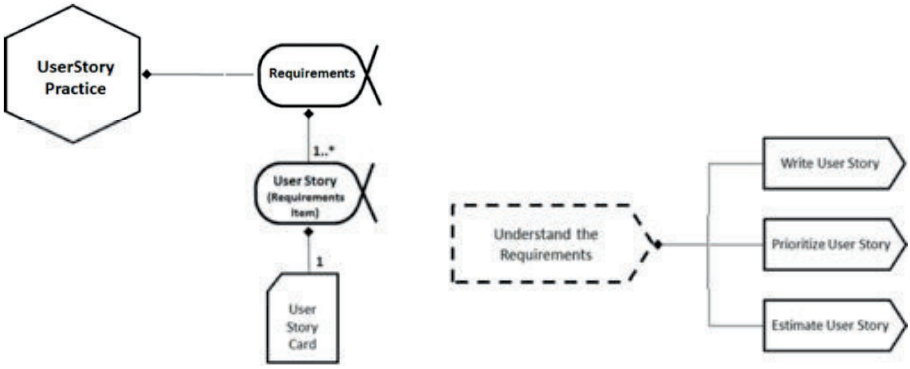


Abbildung 4: Beispiele für die Modellierung von User Stories.

jeder Activity Space mit mindestens einer Aktivität belegt wurde. Auch eine Überprüfung, ob eine Methode alle Alphas des Kernels abdeckt ist möglich, um so Hinweise darauf zu erhalten, ob eine Methode als vollständig erachtet werden kann oder weitere Practices hinzugefügt werden müssen, wenn alle Aspekte abgedeckt werden sollen.

4 Der Einsatz von ESSENCE

Eine der Zielsetzungen von ESSENCE ist es, alle „Lebensphasen“ von Softwareentwicklungsmethoden abzudecken, d.h. ihre initiale Beschreibung, ihre Komposition und Adaption und ihre Ausführung im Projekt. Für einige ausgewählte Beispiele wird dies im Folgenden diskutiert.

4.1 Modellierungsbeispiele

Im modellierungstechnischen Sinne bilden Kernel, Practices und Methoden Instanzen des Metamodells der ESSENCE-Sprache. Bereits die Beschreibung eines einzelnen Alphas, Work Products oder einer einzelnen Aktivität stellt damit ein gültiges, minimales Modell in ESSENCE dar. Auch der in Abbildung 3 bereits gezeigte Überblick über den Kernel ist (abgesehen von der farbigen Hinterlegung der drei Teilbereiche) ein gültiges Modell. Abbildung 4 zeigt einige Modellierungsbeispiele, die aus der ESSENCE-Spezifikation entnommen sind und die gemeinsam das Vorgehen bei der Anforderungserhebung mit User Stories modellieren. Es ist leicht zu erkennen, dass diese Modellierung in der hier gezeigten Form nicht vollständig ist, da den einzelnen Elemente keine detaillierteren Beschreibungen zugewiesen sind. Für diese führt ESSENCE die im Folgenden erläuterte Karten-Metapher ein.

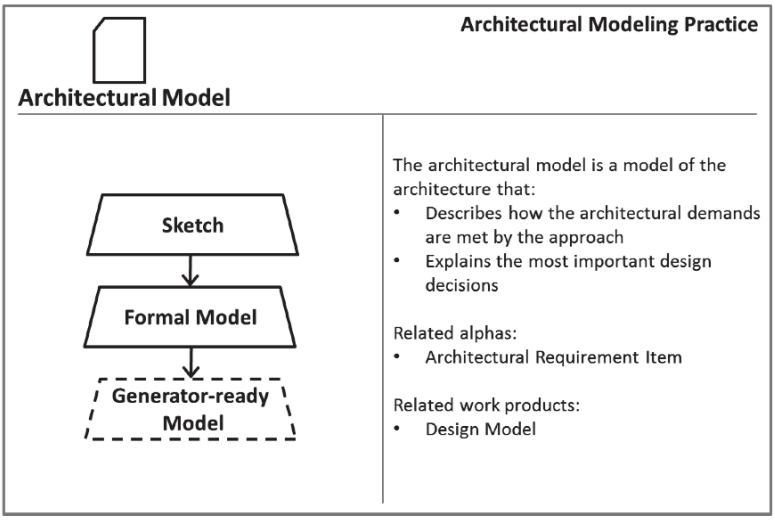


Abbildung 5: Kartenlayout für die Beschreibung eines Work Products (aus *ad/2012-11-01*).

4.2 Karten-Metapher

Ergänzend zur Modellierung von Methoden in Diagrammen führt ESSENCE zur geeigneten Präsentation von Details der Modellelemente die Metapher von Karten im Sinne von Kartei- oder Spielkarten ein. Ob diese Karten physisch vorliegen oder in einem Softwarewerkzeug virutell repräsentiert werden, ist für den Ansatz unerheblich. Die ESSENCE-Spezifikation definiert für jedes der in Abbildung 2 gezeigten Elemente des Metamodells ein Layout für eine Karte, mit der dieses Element dokumentiert werden kann. Abbildung 5 zeigt ein Beispiel für eine solche Karte, die in diesem Fall ein Work Product beschreibt. Jede Practice kann so durch einen Kartensatz repräsentiert werden. Die Komposition von Practices zu Methoden entspricht dann dem Zusammenstellen eines großen Kartensatzes aus mehreren kleinen Sätzen, wobei auch der Kernel durch einen Kartensatz repräsentiert werden kann. Die notwendige Zuordnung von Elementen zueinander kann dabei physisch und kollaborativ in einer größeren Gruppe von Beteiligten geschehen und ist damit nicht auf die Nutzung eines spezialisierten Softwarewerkzeugs beschränkt. Analog kann die Beobachtung eines laufenden Projektes ebenfalls durch die Verwendung von Karten (insbesondere für die Alpha States) geschehen, die auf einem Tableau (z.B. einem Kanban-Board [Epp11]) verschoben werden, um den aktuellen Zustand anzuzeigen. Details zum Enactment folgen weiter unten in diesem Artikel.

Auch außerhalb des eigentlichen Einsatzes im Projekt und in der Erstellung und Dokumentation von Methoden können die Karten eingesetzt werden: Sie bilden zum Beispiel in sich abgeschlossene Einheiten, die als Lerneinheiten sowohl in unternehmensspezifischen Trainee-Programmen als auch im allgemeinen universitären Curriculum Anwendung finden können. Genauso, wie aus einzelnen Karten Kernel, Practices und Methoden zusam-

mengestellt werden können, können sie auch für Lehrveranstaltungen und Kurse thematisch gruppiert werden und einen roten Faden bilden, der den Stoff in handliche Stücke zerlegt.

4.3 Enactment

Die Semantik von ESSENCE zur Ausführungszeit einer Methode basiert auf drei Aspekten: Der Feststellung des aktuellen Projektzustandes in Form von Alpha States, der Erzeugung von Handlungsempfehlungen in Form von Activities sowie der Möglichkeit der Adaption durch Komposition und Erweiterung.

Die Feststellung des aktuellen Projektzustandes erfolgt anhand der Checkpoints, die mit jedem Alpha State verknüpft sind. Unabhängig von der Verwendung in Methoden können diese Checkpoints zu jedem beliebigen Zeitpunkt ausgewertet werden, um eindeutig festzustellen, ob sich eine Instanz eines Alphas in einem bestimmten Alpha State befindet oder nicht. Dabei wird davon ausgegangen, dass jede Instanz eines Alphas seinen Zustand zu jedem beliebigen Zeitpunkt wechseln kann und an keine Reihenfolge der Zustände gebunden ist. Ebenso können zu jedem Zeitpunkt neue Instanzen eines Alphas in das Projekt eintreten oder es verlassen.

Ist der aktuelle Zustand eines Projektes bekannt, kann dieser mit dem gewünschten Zustand verglichen werden. Da die Alpha States geordnet sind, d.h. einen eindeutigen Vorgänger und Nachfolger haben, kann aus diesen Informationen abgeleitet werden, welche Zustände gegebenenfalls zusätzlich durchlaufen werden sollten, um vom aktuellen Zustand in den gewünschten zu gelangen. Aus dieser Zustandsmenge heraus kann dann wiederum eine Menge von Aktivitäten abgeleitet werden, die in der Erreichung dieser Zustände resultieren sollten. Diese bilden eine konkrete Handlungsempfehlung. Dabei ist insbesondere zu berücksichtigen, dass nicht automatisch Instanzen von Aktivitäten erzeugt werden, d.h. nicht davon ausgegangen wird, dass alle empfohlenen Aktivitäten auch tatsächlich sofort durchgeführt werden. Ebenso wenig wird aus der Beendigung einer Aktivität automatisch geschlossen, dass nun der angestrebte Alpha State auch erreicht ist. Die Rückkopplung geschieht ausschließlich über die Checkpoints der Alpha States. Dadurch deckt das Enactment in ESSENCE auch die Fälle ab, in denen eine Aktivität abgebrochen werden kann oder muss, weil der angestrebte Zustand aufgrund anderer Umstände bereits erreicht wurde oder ein Rückschritt eingetreten ist, der andere Aktivitäten wichtiger macht.

Dieser flexible Umgang mit Handlungsempfehlungen ist auch Teil des Adaptionskonzeptes von ESSENCE. Das Modell der Entwicklungsmethode darf zur Ausführungszeit jederzeit geändert werden, so dass in derselben Situation möglicherweise unterschiedliche Handlungsempfehlungen gegeben werden, wenn zwischenzeitlich eine Adaption des Modells erfolgt ist. Diese Adaption kann mit denselben Techniken der Komposition und Erweiterung von Practices durchgeführt werden, wie zum Zeitpunkt der Erstellung von Methoden. Dadurch ist es insbesondere möglich, dass die Beschreibungen von Alphas und Alpha States geändert und erweitert werden können, ohne dass deren Instanzen ungültig werden.

5 Erfahrungen und Ausblick

Die Entwicklung von ESSENCE basiert auf mehrjährigen Erfahrungen in industriellen Projekten, aus denen die ESSENCE-Spezifikation mehrere Beispiele enthält, in denen vorhandene Vorgehensmodelle erfolgreich in ESSENCE modelliert wurden. Ferner wird dort an Beispielen gezeigt, wie aus einer Menge von Practices auf Basis des Kernels Entwicklungsmethoden für verschiedene Projektkategorien (z.B. Prototypprojekte oder Maintenance-Projekte) zusammengestellt werden, wobei in den verschiedenen Methoden unterschiedliche Aspekte fokussiert werden.

Die SEMAT-Initiative betreibt derzeit (Stand Januar 2013) die Standardisierung von ESSENCE im Rahmen der OMG. Die reine technische Dokumentation wird außerhalb des Standards durch Publikationen [JNM+13] und Workshops [SWE12] ergänzt.

Literatur

[Epp11] Thomas Epping. *Kanban für die Softwareentwicklung*. Informatik im Fokus. Springer Berlin Heidelberg, 2011.

[ESMB12] Brian Elvesaeter, Michael Striewe, Ashley McNeile und Arne-Jorgen Berre. Towards an Agile Foundation for the Creation and Enactment of Software Engineering Methods: The SEMAT Approach. In *Second Workshop on Process-based approaches for Model-Driven Engineering (PMDE 2012)*, 2012.

[HSGP05] Brian Henderson-Sellers und Cesar Gonzalez-Perez. The rationale of powertype-based metamodelling to underpin software development methodologies. In *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling - Volume 43*, APCCM '05, Seiten 7–16, 2005.

[ISO07] Software Engineering – Metamodel for Development Methodologies, ISO/IEC 24744. International Organization for Standardisation (ISO), February 2007.

[JNM+13] Ivar Jacobson, Pan-Wei Ng, Paul E. McMahon, Ian Spence und Svante Lidman. *The Essence of Software Engineering: Applying the SEMAT Kernel*. Addison-Wesley Professional, 2013.

[KMSG+12] Mira Kajko-Mattsson, Michael Striewe, Michael Goedicke, Ivar Jacobson, Ian Spence, Shihong Huang, Paul McMahon, Bruce MacIsaac, Brian Elvesater, Arne J. Berre und Ed Seymour. Refounding software engineering: The Semat initiative (Invited presentation). In Martin Glinz, Gail C. Murphy und Mauro Pezzè, Hrsg., *ICSE*, Seiten 1649–1650. IEEE, 2012.

[SPE08] Software & Systems Process Engineering Metamodel Specification (SPEM) Version 2.0, Document formal/2008-04-01. Object Management Group (OMG), April 2008. `http://www.omg.org/spec/SPEM/2.0/`.

[SWE] IEEE Software Engineering Body of Knowledge, Version 3. `http://www.computer.org/portal/web/swebok`.

[SWE12] SEMAT Workshop on Evaluation of Essence, 2012. `http://semat.org/?page_id=678`.

How to Select a Suitable Tool for a Software Development Project: Three Case Studies and the Lessons Learned

Mark Kibanov, Dominik J. Erdmann, Martin Atzmueller
Knowledge and Data Engineering Group, University of Kassel, Germany
{kibanov, erdmann, atzmueller}@cs.uni-kassel.de

Abstract: This paper describes a framework for evaluating and selecting *suitable* software tools for a software project, which is easily extendable depending on needs of the project. For an evaluation, we applied the presented framework in three different projects. These projects use different software development methods (from classical models to Scrum) in different environments (industry and academia). We discuss our experiences and the lessons learned.

1 Introduction

With the growth of the software industry the number of software products (programs, tools, frameworks) with similar functions has also increased. Therefore, the process of selection of the required software has also become more complex. In this paper, we introduce a general three-step framework for selecting suitable software for the current project and environment. Furthermore, we describe the application of the framework to three different software projects: These use different software development methods and environments. All three case studies show promising results and indicate the possibility to apply the suggested framework for a wide range of different projects. In these contexts, we discuss our experiences and the lessons learned.

The rest of the paper is structured as follows: Section 2 discusses related work. After that, Section 3 presents the framework, its advantages, disadvantages and the three distinct steps of selecting the software tools. Section 4 describes three case studies where we applied the framework and the results we obtained during these case studies.

2 Related Work

Starting in 1980 with [Saa80], Thomas L. Saaty developed the AHP (Analytic Hierarchy Process): This method makes it easier for teams to determine the relevant criteria. The Cost-Benefit Analysis is a widely used approach in different disciplines, cf. [Sen00]. General IT-management approaches such as the Capability Maturity Model (CMM) [coo02] and Control Objectives for Information and Related Technology (COBIT) framework [IT 07] describe software acquisition, among other processes. Kneupe [Kne09] investi-

gates the optimization of software acquisition when using the CMMI model - an extension of CMM. Wiese [Wie98] describes the software selection process from an economical point of view. Searching for standard software Wiese uses benefit analysis and describes its efficiency. Nelson et al. [NRS96] describe the software acquisition mainly as a process of making a choice between custom and package software and between insource and outsource acquisition team. Gronau [Gro01] discusses how software is researched, evaluated and finally integrated in large companies. Using two case scenarios, Gronau gives a realistic overview of critical factors for success [Gro12] .

All these approaches are conceptual and describe mainly the process of software acquisition for enterprise, not concentrating on practical issues of selecting particular software for smaller teams (7 - 15 persons). Litke and Pelletier [LP02] describe the Spreadsheet Analysis method for selecting between two alternatives – whether to buy or to build the software. Laakso and Niemi [LN08] use this approach to evaluate different AJAX-enabled Java-frameworks. The method is extended to a scenario-based evaluation similar to our proposed approach.

In comparison to the approaches discussed above, the main difference between methods and the suggested framework is the possibility to apply the framework for application cases without implementing big IT governance system frameworks, which are often rather hard to implement for small teams. The suggested framework can be seen as a modification of the Analytic Hierarchy Process adopted for environments mentioned above.

3 Framework

The proposed framework consists of three main steps:

1. *Identify the software according to minimal and desired requirements*: This results in the list of requirements and software which should be considered in the next steps.
2. *Quantify the requirements*: In order to produce a list of granulated requirements ranked according to their their importance. This list describes the most suitable software product for the project.
3. *Evaluate the software according to the requirements*, i. e., evaluate if the considered software tools meet the requirements.

The result of execution of these three steps is the ranking which shows which software fits to the current project and environment best.

3.1 Step 1: Minimal and Desired Requirements

First, the minimal and desired requirements for the software should be defined by project stakeholders and technical managers so both functional and technical requirements are taken into consideration. Also the users of the software should be interviewed. The person(s) who lead(s) the evaluation may also analyze and add requirements. Stakeholders

include project managers, executives and customers. In this context, the stakeholders make the final decision about the choice of a tool. Minimal requirements may be categorized as operational (which support existing workflows and processes) and technical (which enable the use of software in the current environment). The desired requirements are the features and properties which can optimize the processes but are not usually critical for the team.

Second, the software type should be defined. This task is not as trivial as it seems. The main challenge here is to detect which parameters of the software may be important. Assume, for example, that the project management tool should selected. A specification like "IT project management tool" does not help much as most of the tools support IT project management. Moreover, it is unclear what is specific about IT projects. But if the project uses Scrum [TN86] it gives a clear image of which software type is needed.

Third, the software for further evaluation according to the minimal requirements needs to be selected. The desired requirements may also be used for selecting the software if the minimal requirements are not enough. The person who makes evaluation needs to keep in mind that the selected software will be tested with the selected scenarios.

3.2 Step 2: Quantification of requirements

The aim of the second step is to rank the requirements by their importance. Also some additional requirements may be found in this step, which should not be critical. In the case that the additional requirements are critical, the first step should be executed once again. First, all the future users should rank how important different qualities and requirements of the software are. Each team member gets a list of all collected requirements (in the first step). He or she should then weight the requirements by their importance. During our case studies we figured out that it is easier for users to estimate how important each single feature is, not considering them in the context of the whole system. For example, it is much easier for the user to estimate how important the security of the software is on a scale of 0 to 10 than estimating "which part does the security play in this software?" where the possible answer could be 18%.

Afterwards, the importance scores of the software are normalized in order to get the weight of each requirement, using the following formula:

$$w(r_i) = \frac{a(r_i)}{\sum_{k=1..n} a(r_k)} ,$$

where $w(r_i)$ is the normalized weight of the requirement i, and $a(r_i)$ is the average weight of the requirement i.

Sometimes a second iteration of this procedure is necessary since some of the users (as mentioned below) may add new requirements which should be estimated by all other users. The result of this step is the ranked list ofall requirements, which can be grouped according to functional or organizational role of these requirements for better understanding of the priorities of the project.

3.3 Step 3: Evaluation of Software

The third step is the evaluation of the software and the application of the spreadsheet-analysis method for the final ranking. The main challenge of this part of the evaluation is getting objective information about particular software tools. Of course, the level of this analysis depends on the quantity and accessibility of the software. It is possible to retrieve the information from independent (online- and offline) sources and from the software documentation; information from commercials should be avoided, instead specific technical documentation can be requested from the software vendors. However, it is hard to compare software and the quality of feature implementations relying only on documentation.

We suggest performing a scenario-based analysis to estimate the quality of the implementation of the software features. First, the individual scenarios need to be defined. The scenarios should enable an evaluation of the features that users marked as important and should be done in the environment similar to the environment where the software will be used. One scenario should help to evaluate many aspects of the software, e.g. "Initial installation of the software" may check quality of software support, usability, operating system compatibility. On the other hand, important requirements should be tested with more than one scenario. E.g. if the support is a critical issue, the tester should try to contact vendors and ask some questions during the scenarios.

Second, the scenarios should be executed and the degree of suitability of the software according to the requirements defined in the steps 1 and 2. Then, a decision analysis spreadsheet suggests to give each software the score from -1.0 to 1.0 where 1.0 means "Alternative fully satisfies business requirement or decision criterion" and -1.0 "Alternative fully dissatisfies business requirement or decision criterion." [LP02] However, also other scales may be applied if they have two main properties the person who evaluates the software should not have difficulties giving scores to the software and vice versa: it should be understandable what the different scores mean.

Third, according to the user rating and software evaluation the results of spreadsheet analysis should be calculated and evaluated. The estimated rankings depend on the chosen scale, the number of requirements and the users who estimated the importance of different requirements. Usually about $2 - 3$ alternatives should be considered further for making the final decision. The framework supports decision making but cannot replace the whole process of reaching the decision. The framework is focused on the technical factors and does not consider economical issues such as software license models or integration costs.

4 Case studies

We applied the proposed framework in three different projects. In the following, we introduce each of the projects, the goals of the software evaluation, and discuss the implications of the introduced method.

4.1 Industry Project/Scrum

Capgemini Deutschland GmbH[1] develops the Manufacturing Execution System (MES) for a german car producer company.[2] MES manages the production of the company and is used as an interface between rather slow business processes, e. g., accounting, which are managed in the Enterprise Resource Planning (ERP-) System, and rather quick local production processes, i. e., local devices in the production, cf. [SRSS09]. The team developing the graphical user interface (GUI) of the system decided to move from the current version control system (VCS) SVN[3] to another modern system in order to optimize their internal processes and the collaboration with the IT department of the customer. We were not only concerned with the control version system: Additionally, we expected the whole software development environment to be changed, as VCS plays very an important role.

The first step involved the minimum requirements. They were defined by observing the day-to-day workflows and interviews with developers and project managers. The team used Scrum – a typical agile development method. The two special points about the project were:

1. Each team uses its own tools, utilities and methods for their module development. It was unclear if we have to offer an optimal solution for the GUI team only or for the all MES product teams.
2. The GUI team collaborates with the IT department of the customer. The customer has some requirements for the software development process and has control over the source code of the system. The challenge was to balance the requirements of the team and those of the customer.

We decided to concentrate on the development processes of the team as it was unclear if the unification of the processes of different teams would be possible. Furthermore, the analysis of all requirements and processes of different teams would be very complex and not possible with the existing resources. We also decided to collect the aggregated requirements from the Capgemini employees to minimize contradictions.

In total, we identified 31 systems which are currently developed. Considering these systems and minimal/desired requirements we left only 10 options to select from. In addition, we identified about 20 different requirements and asked all the team members to weight these requirements and to add their own, if necessary. 25 requirements were identified and afterwards estimated by the team members. All the features were divided into five groups:

- Software quality
- Software features
- Integrability to the project environment
- Integrability to the project processes
- Other project relevant properties

The rank of the requirement groups (Table 1) shows that the project-specific features are more important: They obtained an overall score of 65% against 35 % of basic qualities

[1] `http://www.capgemini.de/`

[2] Due to legal restrictions the name of the system and the car producer company cannot be named in this paper.

[3] `http://subversion.apache.org`

Criterion	relative Weight
Basic Qualities	18%
Basic Features	17%
Integrability to the project Environment	22%
Integrability to the development process	34%
Other features relevant for the team	9%

VCS	DAS-Score
Mercurial	77%
Bazaar	74%
Plastic SCM	69%
Git	67%
Synergy	63,5%
Perforce	50%
PureCM	43%
Integrity	40%
AccuRev SCM	39,5%
SVN	38%

Table 1: Weights of different requirements and decision analysis spreadsheet scores for the top 10 version control systems selected for the integration into Capgemini MES Project.

and features. Next, we developed and executed five scenarios, so each of the ten tools was evaluated. The top four tools (Table 1) were considered by the stakeholders of the project. The stakeholders decided to make the transition to Git[4]. The two main factors influenced this decision: some members of the team had previous experience with Git, thus the whole integration seemed to be easier and the customer who could also influence a decision opted for a more "prominent" system like Git. Further details and a full description of this case study can be found in [Kib12].

4.2 Ubiquitous Platform/VENUS

UBICON[5] is the platform and framework for different ubiquitous applications which allows the observation of different social and physical activities [ABD+12]. The platform is developed by Knowledge and Data Engineering group (KDE) at the University of Kassel. The platform is currently hosting the following applications:

- Conferator [ABD+11] – a social conference management system.[6]
- MyGroup – a social system for supporting members of working group.[7]
- WideNoise – the web application for aggregation and illustration of noise-related data collected by the WideNoise smartphone application.[8]
- AirProbe – the web application for case studies measuring air pollution.[9]

The developer team of the UBICON project partially applies the Venus-method for software development [GLRS12] which lets scientists and experts from different areas (e. g.,

[4] http://git-scm.com
[5] http://ubicon.eu/
[6] http://www.conferator.org
[7] http://ubicon.eu/about/mygroup
[8] http://cs.everyaware.eu/event/widenoise
[9] http://cs.everyaware.eu/event/airprobe

Criterion	relative Weight UBICON	relative Weight BibSonomy
Issue Tracker	16,2%	15,8%
Continuous Integration Interface	8,1%	7,3%
User Administration	5,9%	5,6%
Software Reliability	12,5%	17,8%
Version Control System	27,8%	31,7%
Project Management	6%	3,2%
Developer Support	23,5%	18,6%

Table 2: Weight of different requirement types in the UBICON and BibSonomy cases.

System	UBICON DAS-Score	BibSonomy DAS-Score
Redmine	80%	88%
Jira	85%	90%
Trac	57%	66&
FusionForge	45%	53%

Table 3: Decision analysis spreadsheet scores for the top 4 project management systems selected for the UBICON and BibSonomy projects.

information systems, law, usability) work together to create the social acceptable ubiquitous applications. The developers currently use Fusion-Forge for the development of UBICON. Suffering from many different bugs, complicated handling and insecurity of the software the KDE unit aims to replace the Fusion-Forge system. During the first step of the evaluation we performed interviews, where we discussed common use of Fusion-Forge to identify the requirements for the system and later asked the researchers to rate them.

We executed the second step, where the scientists and students estimated how important different requirements are. The results show how the current (and future) software is used and point to some interesting findings. Seven groups of requirements were identified (cf. Table 2). The three biggest groups (Issue Tracker, Version Control System and Developers Support) constitute together 67,5 % of all requirements. Software Reliability (receiving 12,5 %) is also very important for the developers. This can be explained by the significant reliance on the versioning system and the importance during the software development process. Also, the researchers demanded support of new version control systems which can be a signal for transition to Git or Mercurial[10].

Redmine[11] and Jira[12] obtain the two top scores (Table 3). Compared to the BibSonomy case study, the different systems obtained slightly lower scores. Possible explanations, include for example, that different applications are hosted on UBICON. Additionally, some of the applications are developed by international projects and the developers have different locations. Furthermore, UBICON uses a new development approach which implies new requirements for the project management.

[10] http://mercurial.selenic.com
[11] http://www.redmine.org
[12] http://www.atlassian.com/de/software/jira/overview

4.3 Research Projects/Waterfall Model

Benz et al. [BHJ+10] present BibSonomy[13] as a social bookmark and publication system. It aims at helping a researcher in finding literature for his daily work. For this purpose, it uses a growing database of bookmarks and literature references. Since it is simple to use, it has a large number of user, building up a tag-related cloud of papers, links and literature for scientific work.

The development model of BibSonomy is a waterfall model which can be modified dependent on current projects and aims of research. The BibSonomy team currently uses the same instance of Fusion Forge as the UBICON team.

After interviews of the researchers we determined, that the needs of the BibSonomy team are very similar to those of the UBICON team, but the second step of analysis could detect two main differences:

- The version control system is more important for the BibSonomy team;
- The developer support is not as much required by BibSonomy team as by the UBICON team;

Despite these differences and the fact that all of examined systems were ranked lower by the UBICON team, the tendences are the same. As the license costs of Jira are to high, Redmine was selected as the new system for project management of both projects.

4.4 Discussion

All three studies show the possibility to apply the framework in rather different environments. We noticed that the first step was rather easy to execute, but the results were not complete (in all three studies new requirements were discovered in the second step). The second step reveals interesting project properties regarding utility of some features (it is especially interesting to find currently lacking but wished features). The third step usually clearly shows the need (or no need) of the new technology or product: if currently used product is ranked high, the transition may be not necessary. On the other hand, the users tend to rank missing features higher, and thus rank current software lower: In the UBICON and BibSonomy case studies, for example, users ranked interface for version control systems which are currently not used (Git and Mercurial) higher than the systems which are currently used (CVS and SVN). The same tendency can be found in the Capgemini case study, where current system was ranked as last. In all three cases the interview partners were highly motivated to change the current software because they missed some important features. They tended to rank these features high, so the software containing these features got better ranking and thus were ranked higher than current tools.

[13] http://www.bibsonomy.org/

5 Conclusions and Future Work

We presented a three step framework for evaluating and selecting software based on the user's utility and workflows on the one hand and (partially) tested software on the other hand. The presented framework (based on the presented case studies) has the following advantages:

- It is quite flexible (the steps maybe expanded or reduced – depending on special issues of the project and of the software we need to choose from).
- As experienced in the case studies, the framework is easy to use.
- The framework provides a quantified result.

However, the framework has also the following limitations:

- The person who performs the evaluation needs to be able to understand complex technical relations, analyze business processes, and evaluate the software.
- We assume that the framework is easy to apply only in IT teams – as the step two needs technical background (e. g., users need to estimate the importance of security of the software or be able to list additional required software features).
- The framework is focused on the technical requirements and so does not consider such economical factors as software cost or different risks. These should be analyzed afterwards individually for each alternative.

In summary, the framework proved to be an easy and useful tool in the three case studies.

For future work, the framework can be extended to consider not only technical and utility factors but also costs of the software and different risks. In this context the "fair" price of software product may be estimated. Another question is if and how the framework can be extended for special types of software and/or environment. The main challenge is keeping the simplicity of the framework and the transparency during the whole evaluation process.

Acknowledgement

We wish to thank Capgemini Deutschland GmbH and Dr. Mirko Streckenbach and Lukas Birn who helped to organize the first case study, and Dr. Michael Ritzschke from the Humboldt University of Berlin who supervised the Capgemini case study.

This work was supported by the VENUS research cluster Research Center for Information System Design (ITeG) at Kassel University and by the EU-project EveryAware.

References

[ABD+11] Martin Atzmueller, Dominik Benz, Stephan Doerfel, Andreas Hotho, Robert Jäschke, Bjoern Elmar Macek, Folke Mitzlaff, Christoph Scholz, and Gerd Stumme. Enhancing Social Interactions at Conferences. *it – Information Technology*, 53(3):101–107, May 2011.

[ABD+12] Martin Atzmueller, Martin Becker, Stephan Doerfel, Mark Kibanov, Andreas Hotho, Björn-Elmar Macek, Folke Mitzlaff, Juergen Mueller, Christoph Scholz, and Gerd Stumme. Ubicon: Observing Social and Physical Activities. In *Proc. 4th IEEE Intl. Conf. on Cyber, Physical and Social Computing (CPSCom 2012)*, 2012.

[BHJ+10] Dominik Benz, Andreas Hotho, Robert Jäschke, Beate Krause, Folke Mitzlaff, Christoph Schmitz, and Gerd Stumme. The Social Bookmark and Publication Management System BibSonomy. *VLDB*, 19(6):849–875, 2010.

[coo02] Software Acquisition Capability Maturity Model (SA-CMM) Version 1.03. Technical report, 2002.

[GLRS12] Kurt Geihs, Jan Marco Leimeister, Alexander Roßnagel, and Ludger Schmidt. On Socio-technical Enablers for Ubiquitous Computing Applications. In *SAINT*, pages 405–408. IEEE, 2012.

[Gro01] Norbert Gronau. *Industrielle Standardsoftware-Auswahl und Einführung*. München, 2001.

[Gro12] Norbert Gronau. *Handbuch der ERP-Auswahl [mit Mustervorlagen auf CD]*. GITO, Berlin, 2012.

[IT 07] IT Governance Institute, editor. *CobiT 4.1: Framework, Control Objectives, Management Guidelines, Maturity Models*. IT Governance Institute, Rolling Meadows, 2007.

[Kib12] Mark Kibanov. Untersuchung von Versionsverwaltungssystemen mit Zielsetzung der Optimierung der kollaborativen Entwicklung. Master's thesis, Humboldt-University of Berlin, 2012.

[Kne09] Ralf Kneuper. Verbesserung der Beschaffung von Produkten und Leistungen auf Basis des CMMI für Akquisition (CMMI-ACQ), 2009.

[LN08] Tuukka Laakso and Joni Niemi. An evaluation of AJAX-enabled java-based web application frameworks. In Gabriele Kotsis, David Taniar, Eric Pardede, and Ismail Khalil Ibrahim, editors, *MoMM*, pages 431–437. ACM, 2008.

[LP02] Christian Litke and Michael Pelletier. Build it or buy it? - For CEOs Tech Cell, 2002.

[NRS96] Paul Nelson, William B. Richmond, and Abraham Seidmann. Two Dimensions of Software Acquisition. *Commun. ACM*, 39(7):29–35, 1996.

[Saa80] Thomas L. Saaty. *The analytic hierarchy process : planning, priority setting, resource allocation*. McGraw-Hill International Book Co., New York; London, 1980.

[Sen00] Amartya Sen. The Discipline of Cost-Benefit Analysis. *The University of Chicago Press*, 29, 2000.

[SRSS09] Michael Schäfer, Jens Reimann, Christoph Schmidtbauer, and Peter Schoner. *MES: Anforderungen, Architektur und Design mit Java, Spring & Co*. Software + Support Verlag, 2009.

[TN86] Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. *Harvard Business Review*, 1986.

[Wie98] Jens Wiese. Ein Entscheidungsmodell für die Auswahl von Standardanwendungssoftware am Beispiel von Warenwirtschaftssystemen. Technical report, 1998.

Ein pragmatischer Ansatz zur Entwicklung situationsgerechter Entwicklungsmethoden

Michael Spijkerman

s-lab – Software Quality Lab
Universität Paderborn
Zukunftsmeile 1
33102 Paderborn
mspijkerman@s-lab.uni-paderborn.de

Abstract: Bei der Erstellung oder Anpassung von Entwicklungsmethoden ist es die Aufgabe des Methodenentwicklers die richtigen Methodenanforderungen zu berücksichtigen. Zum einen gibt es Methodenanforderungen zur Optimierung der Entwicklungsmethode. Zum anderen gibt es situative Methodenanforderungen, welche die Situation, in der die Entwicklungsmethode ausgeführt wird, in Betracht ziehen. Eine Situation wird beschrieben über Rahmenbedingungen und Eigenschaften, die Situationsfaktoren genannt werden und Einfluss auf die Entwicklungsmethode haben. Die Berücksichtigung der Situationsfaktoren ist notwendig, da sonst eine Entwicklungsmethode im Hinblick auf die zu erreichenden Optimierungen erstellt wird, aber Schwierigkeiten bei der Einführung und Benutzung mit sich bringen kann. Das Forschungsgebiet des Situational Method Engineering (SME) erforscht die Entwicklung situationsgerechter Entwicklungsmethoden.
In diesem Artikel werden Schwierigkeiten in Industrieprojekten beschrieben, die aus der fehlenden Berücksichtigung der Situationsfaktoren resultieren und die bei Anwendung von SME-Ansätzen im industriellen Umfeld entstehen. Zur Lösung dieser Schwierigkeiten wird eine Checkliste von möglichen Situationsfaktoren und weiterhin ein pragmatisches Vorgehen zur Ermittlung von Methodenanforderungen, welches die situativen Methodenanforderungen berücksichtigt, vorgestellt.

1 Einleitung

In Projekten im s-lab – Software Quality Lab[1] erkennen wir oft die Notwendigkeit Entwicklungsmethoden neu zu erstellen oder implizit vorhandene Entwicklungsmethoden explizit festzulegen. Das bildet die Grundlage mit einer strukturierten Herangehensweise Optimierungen im Hinblick auf die Qualität des Zielprodukts zu erreichen. Das bedeutet die Einführung von Artefakten oder die Definition von Prozessen und Aktivitäten als Voraussetzung für konstruktive oder analytische Qualitätssicherungsmaßnahmen. Dabei ist es besonders wichtig Rahmenbedingungen und Einflüsse auf die Entwicklungsmethode zu betrachten, damit verschiedene Probleme verhindert werden können.

[1] www.s-lab.de

Exemplarisch werden im Folgenden drei Beispiele für eine Fehlplanung bei der Methodenentwicklung dargestellt. In einen Projekt sollte der Entwicklungsprozess durch ein Modellierungswerkzeug unterstützt werden, mit dem die Erstellung formalisierter Spezifikationsinhalte vereinfacht wird. Nach der Planung und Evaluierung möglicher Werkzeuge war der Kauf oder die Entwicklung des passenden Modellierungswerkzeugs nicht möglich. Im Unternehmen wurde kein Nutzen über das Projekt hinaus erkannt und die Investition wurde nicht getätigt. In einem anderen Beispiel war die Beschreibung eines Entwicklungsartefakts mit einer formalisierten Sprache angedacht, doch die Anwendung der Sprache war durch mangelnde Kenntnisse und Verständnis der Sprache hinderlich bei der Erstellung des Artefakts. In einem weiteren Beispiel gab es Ressourcenprobleme bei der Entwicklung von zusätzlicher Softwareschnittstellen, da die Notwendigkeit bestand standardisierte, etablierte Softwareschnittstellen zusätzlich anzubieten zu müssen. In allen drei fällen hätten die situativen Eigenschaften Kosten, Know-how, Ressourcenknappheit und Berücksichtigung bestehender Standards im Vorfeld berücksichtigt werden können.

Die Erkenntnis aus diesen Problemen ist, dass eine optimale Lösung im Hinblick auf die Qualität des Entwicklungsziels nicht immer erreicht werden kann. Es sollte vielmehr eine passende Lösung im Hinblick auf die Anwendbarkeit angestrebt werden.

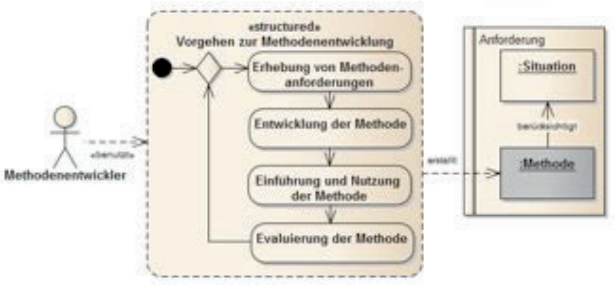


Abbildung 1: Das zugrunde liegende Vorgehen mit der Anforderung "Die Entwicklungsmethode soll die gegebene Situation berücksichtigen"

Auf Basis dieser Erkenntnis ist das Forschungsgebiet der situationsgerechten Methodenentwicklung (engl. Situational Method Engineering (SME)) entstanden (vgl. [HSR10]), bei dem Entwicklungsmethoden unter Berücksichtigung der Situation entwickelt werden. In diesem Artikel wird davon ausgegangen, dass das Vorgehen der Methodenentwicklung die Schritte *Erhebung von Methodenanforderungen* und *Entwicklung der Methoden*, neben Weiteren, umfasst. SME beschreibt die Anforderung, dass die erstellte Methode die Situation berücksichtigen soll (vgl. Abbildung 1).

Die angedachte Lösung zur Umsetzung der in Abbildung 1 beschriebenen Anforderung ist die Ermittlung von Methodenanforderungen. Auf diese Weise können die zu erreichenden Ziele bei einem Methodenentwicklungsprojekt frühzeitig festgelegt und mit den Stakeholdern abgeglichen werden. Damit zusätzlich die Situation der Methode, entsprechend in situativen Methodenanforderungen, berücksichtigt werden kann muss zuerst die Situation der Entwicklungsmethode ermittelt werden. Das kann mit Hilfe einer Checkliste aller möglichen Situationsfaktoren geschehen, auf dessen Basis die gültigen Situa-

tionsfaktoren für das Methodenentwicklungsprojekt ermittelt werden. Aus den gültigen Situationsfaktoren können situative Methodenanforderungen erstellt werden. Sind Methodenanforderungen definiert, vergleichbar mit Anforderungen in Softwareentwicklungsprozessen, können sie analysiert und verfeinert werden, damit sie in der Entwurfsphase geeignet respektiert werden können (vgl. Abbildung 2).

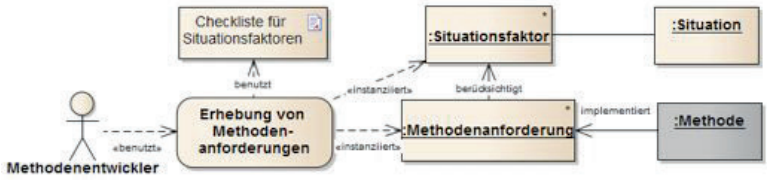


Abbildung 2: Lösungsidee

Die vorgeschlagene Lösung umfasst zwei Schwierigkeiten. Zum einen es gibt keine Checkliste aller möglichen Situationsfaktoren. Zum anderen fehlt ein passendes Vorgehen zur Ermittlung von Methodenanforderungen.

1.1 Schwierigkeiten bei der Ermittlung der Situation

Für den Begriff „Situation“ gibt es keine eindeutige Definition, wie in [BKKW07] diskutiert wird. Eine Auswahl der Konzepte, die für die Beschreibung von situativen Eigenschaften genutzt werden, heißen *Reference Context*, *Project Environment*, *Project Situation*, *Development Situation*, *Project Environment* oder *Project Context*. Der gemeinsame Aspekt dieser Konzepte ist die Beschreibung von Einflussfaktoren auf die zu erstellende Methode. Nachvollziehbar ist hier die Bestimmung des *Project Context* mit den 17 *contingency factors* von [SH96], die in [KDS07] um weitere Erkenntnisse vervollständigt werden. In einer anderen Arbeit [BWBM08] werden andere, empirisch ermittelte, Einflussfaktoren auf eine Entwicklungsmethode unter dem Namen Situationsfaktoren vorgestellt. In diesem Artikel übernehmen wir die zuletzt genannten Begrifflichkeiten: „eine *Situation* wird die über *Situationsfaktoren* definiert“.

Wir erkennen die Schwierigkeit einer vollständigen Situationsbestimmung, denn keine der vorhandenen Listen [KDS07] und [BWBM08] ist vollständig. [Coc00] geht zusätzlich auf menschliche Aspekte ein. [HSR10] beschreibt zusätzlich Normen und Standards, die Auswirkungen auf einzelne Methodenelemente haben. Vertragseigenschaften sind nach [CS05] und eingesetzte Werkzeuge nach [KA04] weitere Aspekte.

1.2 Schwierigkeiten bei der Ermittlung von Methodenanforderungen

Ähnlich zu Softwareentwicklungsprozessen weiß der Entwickler ohne vorhandene Anforderungen nicht was er implementieren soll. Demnach müssen auch Methodenanforderungen erhoben werden.

In [Ral02] werden Strategien zur Ermittlung von Methodenanforderungen beschrieben. Strategien zur Erhebung von Methodenanforderungen bei einer Methodenneuentwick-

lung und Methodenanpassung werden unterschieden. Bei der Methodenanpassung werden Anforderungen der ursprünglichen Methode auf bestehende Gültigkeit hin überprüft, bestehende Anforderungen erweitert oder neue hinzugefügt. Hier fehlt eine Anleitung wie genau eine einzelne Anforderung ermittelt werden kann. Bei der Methodenneuentwicklung gibt die *Activity-driven Strategy* den konkreten Hinweis notwendige Aktivitäten der Methode zu ermitteln. Mit Hilfe der notwendigen Aktivitäten können Anforderungen an einen möglichen Entwicklungsprozess abgeleitet werden. In beiden Fällen muss der Methodenentwickler sich auf sein Methodenverständnis und seine Kreativität verlassen, um relevante Anforderungen zu erheben.

Der Ansatz nach [Ral02] führt zusätzlich zur Umsetzung der Strategien eine formalisierte Sprache ein. Obwohl diese Sprache für Methodenentwickler mit Literaturverständnis über SME intuitiv und bekannt ist, waren im Industrieprojekt bei den beteiligten Mitarbeitern Schwierigkeiten vorhanden die Sprache zu verstehen und anzuwenden. Weitere Ansätze zur Ermittlung von Methodenanforderungen sind nicht bekannt.

1.3 Aufbau des Artikels

Die Problemstellung in Kapitel 1 beschreibt die Erkenntnis situative Methodenanforderungen zu erheben. Jedoch sind Schwierigkeiten bei der Ermittlung der Situation vorhanden. Darüberhinaus gibt es in der Literatur kein passendes Vorgehen zur Erhebung von Methodenanforderungen.

In Kapitel 2 wird zunächst eine Checkliste möglicher Situationsfaktoren vorgestellt, die mittels einer umfassenden Literaturrecherche ermittelt wurde. Diese Checkliste kann anschließend von einem Methodenentwickler genutzt werden, um umsichtig alle bekannten Situationsfaktoren zu begutachten und die für sein Methodenentwicklungsprojekt gültigen Situationsfaktoren zu ermitteln.

Danach wird ein, durch ein Industrieprojekt motiviertes, pragmatisches Vorgehen zur Erhebung von Methodenanforderungen in Kapitel 3 beschrieben. Es ersetzt das mit Schwachstellen behaftete Vorgehen nach [Ral02]. Weiterhin ermöglicht es die Erkenntnisse über die Situation als Methodenanforderung in den Entwicklungsprozess der Methode einfließen zu lassen.

Dieser Artikel schließt mit der Zusammenfassung und Ausblick in Kapitel 4.

2 Beschreibung einer Situation

Damit eine Situation beschrieben werden kann, müssen alle Situationsfaktoren berücksichtigt werden. Bei dieser Aufgabe hilft eine konsolidierte Checkliste aller bekannten, möglichen Situationsfaktoren. Ansonsten können Einflussfaktoren vergessen werden. Diese Checkliste hilft dem Methodenentwickler die für sein Methodenentwicklungsprojekt gültigen Einflussfaktoren zu ermitteln. Im Folgenden wird die Herangehensweise zur Erstellung der Checkliste dargestellt. Im Anschluss wird die Checkliste vorgestellt.

In der Literatur (bspw. [KDS07]) wird der Fokus für eine Situation oft auf Projekteigenschaften gelegt. Erfahrungen in der praktischen Arbeit im Industrieprojekt zeigen weitere Schwierigkeiten. Beispielsweise bedingen Eigenschaften des Unternehmens oder der Entwicklungsdomäne die Erstellung und Einführung von Entwicklungsmethoden.

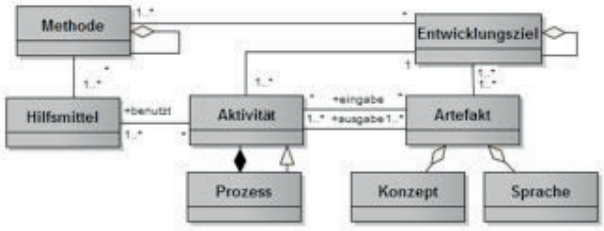


Abbildung 3: Angepasste Definition von Methodenelementen nach [Sau11]

Das Themengebiet SME identifiziert neben Projekt-, Unternehmens- und Domäneneigenschaften weitere Einflussfaktoren, wie *Auftraggeber-Auftragnehmer-Beziehungen*, *Eigenschaften des Entwicklungsvorhabens*, *Eigenschaften des Kunden*, *Menschliche Eigenschaften* und allgemeinere *Markt- und Produkteigenschaften*. Es gibt Hinweise darauf, dass Situationsfaktoren Auswirkungen auf einzelne Methodenelemente haben [KA04].

Kategorie	Unterkategorie
Eigenschaften des Projekts	
Eigenschaften der Organisation	
Eigenschaften der Domäne	
Menschliche Eigenschaften	Stakeholder Eigenschaften der Entwickler
Eigenschaften des Entwicklungsvorhabens	Wiederverwendungsstrategie Realisierungsstrategie Entwicklungsvorhabens Einführungsstrategie Entwicklungsstrategie
Eigenschaften der Kunden	
Auftraggeber-Auftragnehmer-Beziehungen	Vertragsgestaltung Geschäftsbeziehungen Kooperationseigenschaften
Markteigenschaften	
Produkteigenschaften	
Rahmenbedingungen für Methodenelemente	Arbeitsprodukte Prozesse Werkzeuge Rollen Aktivitäten Meilensteine (Teil-)Methoden

Kategorie	Unterkategorie	Situationsfaktor
Eigenschaften der Domäne	Systemklasse	Datenbanksystem Dynamisches System Eingebettetes System Selbst-Adaptives System Real-Zeit-Sytem GUI-basiertes System
	Qualitätseigenschaften	Formalität Beziehungen Abhängigkeiten Komplexität Wiederholbarkeit Variabilität Variable Artefakte Sicherheit Ausfallsicherheit Real-Zeit Kritikalitätssicherheit
	Arbeitsprodukte der Domäne	
	Techniken und Paradigmen	SOA Produktlinien

Abbildung 4: Checkliste mit Kategorie, Unterkategorie und Beispiel möglicher Situationsfaktoren für "Eigenschaften der Domäne"

Zur besseren Strukturierung der Situationsfaktoren werden Kategorien mit Unterkategorien eingeführt. Beispielsweise unterhalb der Kategorie *Rahmenbedingungen für Methodenelemente* werden die grundlegenden Methodenelemente als Unterkategorien aufgeführt. Die Abbildung 1 zeigt die angepasste Definition eines Methodenmetamodells nach [Sau11]. Eine Methode betrachtet immer ein oder mehrere Entwicklungsziele. Das Ent-

wicklungsziel bedingt Artefakte, die aus einem Konzept und einer Sprache zur Beschreibung des Konzepts bestehen sowie Aktivitäten, die Artefakte erstellen. Mehrere Aktivitäten können über einen Prozess in eine definierte Reihenfolge gebracht werden. Letztendlich beschreibt die Methode Hilfsmittel zur Durchführung der Aktivitäten.

In Abbildung 4 werden die Kategorien, Unterkategorien und exemplarisch die möglichen Situationsfaktoren der Kategorie *Eigenschaften der Domäne* vorgestellt.

Nicht alle Situationsfaktoren können in der Checkliste vorgegeben werden. Beispielsweise ist es nicht realistisch mögliche Situationsfaktoren für die Unterkategorie *Arbeitsprodukte der Domäne* in einer Checkliste zu beschreiben, da Expertenwissen aller Domänen erforderlich wäre. Bei der Erhebung dieser Informationen muss ein Domänenexperte beim aktuellen Methodenentwicklungsprojekt unterstützen.

Die Checkliste weist auf mögliche Situationsfaktoren hin, die konkrete Ermittlung gültiger Situationsfaktoren bleibt Aufgabe des Methodenentwicklers.

3 Erhebung von Methodenanforderungen

In Abschnitt 1.2 wurden Schwachstellen bei existierenden Ansätzen erläutert. Die Anwendbarkeit in einem Industrieprojekt war nicht gegeben. Mangelnde Fähigkeit im Umgang mit vorgegebenen Sprachen, fehlende konkrete Anweisungen zur Anforderungserhebung und nicht umfassende Berücksichtigung von Methodenelementen waren Ursachen.

Ein pragmatischeres Vorgehen zur Erhebung von Methodenanforderungen ist notwendig. Im selben Industrieprojekt wurde ein Vorgehen zur Erhebung Anforderungen im Bereich der Systementwicklung genutzt. Das Vorgehen auf Basis der allgemeinen Erkenntnissen der Requirements Engineering (bspw. [Rup09] und [RR99]), sowie dem für die Systementwicklung spezifischen SYSMOD Vorgehen aus [Wei08], umfasst die Schritte *Definieren von Zielen*, *Erstellung eines Kontextdiagramms*, *Definition der Stakeholder*, *Erstellung eines Glossars*, *Ermittlung funktionaler Anforderungen* und *Ermittlung nicht-funktionaler Anforderungen*. Das Vorgehen hat sich als praktikabel erwiesen und wurde gut angenommen. Die positive Erfahrung ist der Auslöser das pragmatische Vorgehen von der Systementwicklung auf die Methodenentwicklung zu übertragen. Bei der Übertragung muss klar werden, was funktionale und nicht-funktionale Anforderungen einer Methode sind.

Funktionale Anforderungen im Softwareentwicklungsprozess beschreiben welche Funktionalitäten von der Software umgesetzt werden sollen. Es stellt sich die Frage „Was soll die Methode umsetzen?“. Zunächst definiert eine Methode Aktivitäten, Artefakte, Prozesse und Hilfsmittel. An dieser Stelle besteht der Anspruch auch die zusätzlichen Elemente bei der Erhebung von Methodenanforderungen zu berücksichtigen, als nur Aktivitäten wie in [Ral02]. Darüberhinaus dokumentiert und regelt die Methode die Entwicklung des Entwicklungsziels. Aus diesem Grund werden die funktionalen Anforderungen als Anforderungen an die Methodenelemente angesehen.

[CL09] sieht nicht-funktionalen Anforderungen als zu erfüllende Rahmenbedingungen des Entwicklungsziels an. Eine sehr abstrakte nicht-funktionale Anforderung an die zu erstellende Methode wird in diesem Artikel hervorgehoben, die Situationsgerechtigkeit. In dem vorangegangen Kapitel wurde die Situation über Situationsfaktoren definiert. Die nicht-funktionalen Anforderungen einer Methode sind Anforderungen, die auf Basis der ermittelten gültigen Situationsfaktoren erstellt werden.

Das pragmatische Vorgehen für die Anforderungserhebung von Methodenanforderungen sieht die folgenden, angepassten Schritte vor:

1) Definieren von Zielen
2) Erstellung eines Kontextdiagramms
3) Definition der Stakeholder
4) Erstellung eines Glossars
5) Ermittlung von Anforderungen an Methodenelemente
6) Ermittlung von Anforderungen aus Situationsfaktoren

1) Definieren von Zielen. Damit keine Unklarheit darüber besteht warum die Anstrengung der Methodenentwicklung geleistet werden soll sind zuerst die Ziele zu beschreiben. Die Notwendigkeit, warum eine Methode definiert, angepasst oder erweitert wird muss klargestellt werden. Mit dem Wissen über das definierte Ziel werden intuitiv die unterschiedlichsten Dringlichkeiten und Umsetzungsanforderungen klar.

Die folgende Auflistung beschreibt eine Übersicht über Ziele und unterschiedliche Gründe für eine Methodenentwicklung oder -anpassung:

- Qualitätssteigerung der zu entwickelnden Produkte
- Verbesserung des Projektmanagements
- Einhaltung von Normen und Standards
- Erhöhung von Maturity Level (bspw.: CMMI)
- Erhöhung des Verständnisses über die Entwicklungsaktivitäten
- Einführung effizienterer Arbeitsweisen
- Umsetzung von Anforderungen von Auftraggebern oder Auftragnehmern

2) Erstellung eines Kontextdiagramms. Eine Methode braucht Eingabeartefakte und produziert Ausgabenartefakte. Es gibt Benutzer der Methode und sie wird von existierende Systemen, Werkzeugen und anderen Hilfsmitteln unterstützt. Eine Entwicklungsmethode stellt oft eine Teilmethode einer übergeordneten Entwicklungsmethode dar. Das Kontextdiagramm soll die genannten Elemente beschreiben und dadurch zu berücksichtigende Schnittstellen verdeutlichen.

3) Definition der Stakeholder. Es gibt verschiedene Stakeholder, die verschiedene Wünsche bezüglich der zu erstellenden Methode haben. Damit alle Wünsche berücksichtigt werden können, müssen die Stakeholder der Methode ermittelt werden. Dazu muss die Frage „Wer hat Interesse an der Methode?“ beantwortet werden.

Neben den in Schritt 2 definierten Benutzern der Methode ist die Auswahl der relevanten Personengruppen ebenso vom dem in Schritt 1 definierten Zielen abhängig. Beispielsweise bringt die Zieldefinition „Verbesserung einer Schnittstellenspezifikation“ die Stakeholder Systemarchitekt, der konkrete Vorschriften zur Erstellung der Spezifikation erhalten möchte, Entwickler, der konkrete Anforderungen an bestimmte Inhalte stellt oder Manager, der das Ziel formuliert hat, hervor. Ein weiteres Beispiel für die Abhängigkeit von der Zieldefinition, ist gegeben, mit dem vom Manager definierten Ziel „Verbesserung der Zusammenarbeit mit dem Zulieferer“. Stakeholder können in dem Fall neben dem Manager, der Einkäufer und die Zulieferer sein. Es kann eine Stakeholderanalyse, beispielsweise aus [HD07], als strukturierte Herangehensweise genutzt werden.

4) Erstellung eines Glossars. Die Erstellung eines Glossars ist für das gemeinsame Verständnis wichtig. Es sollten die folgenden Begriffe detailliert werden:

- Umfang der Methode: Was ist in der Methode zu berücksichtigen?
- Teilmethoden: Wie wird die Methode in verschiedene Teil-Methoden gegliedert und wie heißen die Teil-Methoden?
- Methodenelemente: Wie werden die einzelnen Elemente der Methode genannt?

5) Ermittlung von Anforderungen an Methodenelemente. Anforderungen an Methodenelemente müssen erhoben werden. Beispielsweise kann die *Activity-driven Strategy* aus [Ral02] wieder aufgegriffen werden (siehe Abschnitt 1.2). Die weiteren Methodenelemente (siehe Abbildung 3) werden ebenfalls in die Betrachtung mit einbezogen. Die Aufgabe der Ermittlung von Methodenelementen wird wie folgt detailliert.

5.1) Ermittlung von notwendigen Artefakten
5.2) Ermittlung von notwendigen Aktivitäten
5.3) Ermittlung von notwendigen Prozessen
5.4) Ermittlung von notwendigen Hilfsmitteln

Momentan gibt es keine formalisierte Herangehensweise für die Durchführung der Aktivitäten (5.1 - 5.4). Eine Möglichkeit ist die detaillierte Analyse der Entwicklungsdomäne. Die Definition der Anforderungen liegt dem gegenüber aber auf einen abstrakteren Level und so ist es ausreichend die wichtigsten und offensichtlichsten Methodenelemente in den Anforderungen zu betrachten. In der Praxis eignen sich dafür Interviews und Workshops mit Domänenexperten.

6) Ermittlung von Anforderungen aus Situationsfaktoren. Gültige Situationsfaktoren beschreiben Einflüsse und Rahmenbedingungen einer Methode. Situationsfaktoren geben noch keine Hinweise darauf wie eine Methode an die Situation angepasst wird. Die Checkliste möglicher Situationsfaktoren soll genutzt werden für die Ermittlung relevanter, gültiger Situationsfaktoren. Danach muss der Methodenentwickler die Situationsfaktoren in zu erfüllende Methodenanforderungen überführen. Beispielsweise können Wünsche der Stakeholder direkt als Methodenanforderungen beschrieben werden. An anderen Stellen muss für jeden gültigen Situationsfaktor der tatsächliche Einfluss auf die Methode analysiert werden. Die Aufgabe der Ermittlung von Anforderungen aus Situationsfaktoren wird wie folgt detailliert.

6.1) Ermittlung gültiger Situationsfaktoren auf Basis der Checkliste
6.2) Analyse des Einflusses der gültigen Situationsfaktoren auf die Methode
6.3) Erstellung von Methodenanforderungen auf Basis der Analyseergebnisse

Die *Entwicklung der Methode* folgt der Aktivität *Erhebung von Methodenforderungen* (vgl. Abbildung 1) und wird momentan auf Basis der Metamethode MetaMe durchgeführt [Sau11]. Werden die situativen Methodenanforderungen bei der Methodenentwicklung implementiert wird eine auf die Situation angepasste Entwicklungsmethode erstellt. Eine *Evaluierung der Methode* soll zukünftig durch eine iterative Ausführung des Vorgehens der Methodenentwicklung erreicht werden.

4 Zusammenfassung

Ausgehend von existierenden Problemen bei der Anwendung von SME-Konzepten im industriellen Umfeld wird eine Lösungsidee dargestellt, die eine umfassende Übersicht möglicher Situationsfaktoren erfordert. Dafür wird in Kapitel 2 eine Checkliste vorgeschlagen.

Im industriellen Kontext wird ein pragmatisches Vorgehen zur Erhebung von Methodenanforderungen gewünscht. Dazu wird in Kapitel 3 ein Vorgehen beschrieben, welches, in dem Schritt *Ermittlung von Anforderungen aus Situationsfaktoren*, explizit auf situative Methodenanforderungen eingeht. Auf diese Weise werden notwendige Methodenanforderungen, welche die gegebene Situation berücksichtigen, in den Methodenentwicklungsprozess eingebracht.

Dieser Ansatz wird momentan in einem Industrieprojekt umgesetzt und zeigt erste Ergebnisse. Eine vollständige Evaluierung steht noch aus. Dieser Ansatz berücksichtigt keine konkreten Sprachen und Vorgaben zur Verwaltung von Anforderungen. An dieser Stelle wird angenommen, dass allgemeine Anforderungswerkzeuge aus der Softwareentwicklung geeignet sind und vom Methodenentwickler genutzt werden können. Sind die Methodenanforderungen erstellt können diese mit allgemeinen Anforderungsanalysen weiter detailliert werden.

Bei der Überführung von Situationsfaktoren in Methodenanforderungen stellt sich das Problem, dass nicht für alle Situationsfaktoren der konkrete Einfluss auf die Methode bekannt und definiert ist. An dieser Stelle müssen detailliertere Anweisungen geliefert werden, so dass für einen ermittelten gültigen Situationsfaktor eine entsprechende Anforderung formuliert werden kann (vgl. Vorgehensschritte 6.2 und 6.3).

Es soll möglich werden konstruktive Anweisungen zur Überführung der Methodenanforderungen in ein Methodendesign zu definieren. Dafür muss ein formalisiertes Prozessmodel zur Definition der Vorgehensschritte und ein Produktmodell zur Festlegung der Typen von Methodenanforderungen erstellt werden. Sobald das hier aufgezeigte Vorgehen formalisiert ist soll es unser bereits formalisiertes Vorgehen zur *Entwicklung der Methode* (MetaMe) erweitern, damit MetaMe situationsgerechte Entwicklungsmethoden erstellen kann.

Literaturverzeichnis

[BKKW07] T. Bucher, M. Klesse, S. Kurpjuweit und R. Winter. Situational Method Engineering: On the Difference of "Context" and "Projekt Type". In J. Ralyté, S. Brinkkemper und B. Henderson-Sellers, Hrsg., Situational Method Engineering: Fundamentals and Experiences, Jgg. 244 of IFIP International Federation for Information Processing, Seiten 33–48. Springer Boston, 2007.

[BWBM08] W. Bekkers, I. van de Weerd, S. Brinkkemper und A. Mahieu. The Influence of Situational Factors in Software Product Management: An Empirical Study. In Proceedings of the 2008 Second International Workshop on Software Product Management, IWSPM '08, Seiten 41–48, Washington and DC and USA, 2008. IEEE Computer Society.

[CL09] L. Chung und J. do Prado Leite. On Non-Functional Requirements in Software Engineering. In A. Borgida, V. Chaudhri, P. Giorgini und E. Yu, Hrsg., Conceptual Modeling: Foundations and Applications, Jgg. 5600 of Lecture Notes in Computer Science, Seiten 363–379. Springer Berlin/Heidelberg, 2009.

[Coc00] A. Cockburn. Selecting a Project's Methodology. IEEE Software, 17(4):64–71, 2000.

[CS05] M. Cossentino und V. Seidita. Composition of a New Process to Meet Agile Needs Using Method Engineering. In R. Choren, A. Garcia, C. Lucena und A. Romanovsky, Hrsg., Software Engineering for Multi-Agent Systems III, Jgg. 3390 of Lecture Notes in Computer Science, Seiten 36–51. Springer Berlin / Heidelberg, 2005.

[HD07] N. Hillebrand und G. Drews. Lexikon der Projektmanagement-Methoden. Haufe Projektmanagement. Haufe-Mediengruppe, 2007.

[HSR10] B. Henderson-Sellers und J. Ralyté. Situational Method Engineering: Stateof-the-Art Review. j-jucs, 16(3):424–478, 2010.

[KA04] F. Karlsson und P. J. Agerfalk. Method configuration: adapting to situational characteristics while creating reusable assets. Information and Software Technology, 46(9):619–633, 2004.

[KDS07] E. Kornyshova, R. Deneckère und C. Salinesi. Method Chunks Selection by Multicriteria Techniques: an Extension of the Assembly-based Approach. In Jolita Ralyté, S. Brinkkemper und B. Henderson-Sellers, Hrsg., Situational Method Engineering: Fundamentals and Experiences, Jgg. 244 of IFIP International Federation for Information Processing, Seiten 64–78. Springer Boston, 2007.

[Ral02] J. Ralyté. Requirements Definition for the Situational Method Engineering. In Proceedings of the IFIP TC8 / WG8.1 Working Conference on Engineering Information Systems in the Internet Context, Seiten 127–152, Deventer and The Netherlands and The Netherlands, 2002. Kluwer, B.V.

[RR99] S. Robertson und J. Robertson. Mastering the requirements process. Addison-Wesley, Harlow, 1999.

[Rup09] C. Rupp. Requirements-Engineering und -Management. Professionelle, iterative Anforderungsanalyse für die Praxis. Carl Hanser Verlag GmbH & CO. KG, München, 5., aktualisierte und erweiterte Auflage, 2009.

[Sau11] S. Sauer. Systematic Development of Model-based Software Engineering Methods. Dissertation, University of Paderborn, Paderborn, 2011.

[SH96] K. van Slooten und B. Hodes. Characterizing IS development projects. In Proceedings of the IFIP TC8, WG8.1/8.2 working conference on method engineering on Method engineering : principles of method construction and tool support: principles of method construction and tool support, Seiten 29–44, London and UK, 1996. Chapman & Hall, Ltd.

[Wei08] T. Weilkiens. Systems Engineering mit SysML/UML: Modellierung, Analyse, Design. dpunkt Verlag, 2., aktualisierte und erweiterte Auflage, 2008.

Assembly-based Method Engineering with Method Patterns

Masud Fazal-Baqaie, Markus Luckey, Gregor Engels

s-lab – Software Quality Lab
Universität Paderborn
Zukunftsmeile 1
33102 Paderborn
{masudf,luckey,engels}@uni-paderborn.de

Abstract: Software development methods prescribe and coordinate the activities necessary to plan, build, and deliver software. To provide methods that account for the situational context of a development project, e.g., an acquirer-supplier-relationship or specific communication needs, the existing method creation approaches represent a trade-off between flexibility and ease of use. On the one side, less flexible configurable methods offer a fixed set of configurations to quickly adapt a method to the situation at hand. On the other side, assembly-based approaches allow creating methods from scratch by combining preexisting building blocks. Thus, they are more flexible and capable of creating methods not covered by configurations of configurable methods, e.g., a mixture of agile and plan-driven ideas. However, assembly-based approaches are not easy to use and require considerable expert knowledge. In this paper we suggest the use of method patterns during the assembly-based method creation. Method patterns represent desirable principles for the to-be-method and therefore support the right choice and combination of method building blocks, simplifying assembly-based method creation.

1. Introduction

Large software development projects often involve many stakeholders and different organizations. One example for such a project is the development of an ePassport system. An ePassport system covers all lifecycle phases of an ePassport, from the data collection during the enrollment of its holder, over its personalization (the "printing") to its delivery to its holder, its usage, and finally its destruction.

In order to successfully accomplish large software projects like ePassport projects, software engineering methods are applied. By software engineering methods we denote the full set of elements needed to describe a software development project, including the development process and its activities, the artifacts produced, and the tools and techniques that are employed as well as relationships between these concepts [ES10].

There exist several widespread software engineering methods based on different philosophies for different purposes, e.g., RUP [Kr99], V-Modell XT [Vm12] or Scrum [SS11]. However, even for one specific domain like the development of ePassport systems there

is no one-size-fits-all method. The very different nature and priorities of each project, i.e., the situational context [HR10], has an impact on the method's activities and artifacts. As an example, consider the trade-off between plan-driven activities and agility [Bo03]. That trade-off is influenced, e.g., by having an acquirer-supplier-relationship, like it is typical for ePassport projects, or by the stability of the requirements base. There may also exist certain regulatory constraints that have to be taken into account, e.g., to meet a certain process maturity level regarding CMMI [Cm10].

Situational method engineering (SME) is the field dedicated to engineering situation-specific software development methods from scratch or adapting existing methods [HR10]. The adaption of existing methods is often summarized under the term tailoring, disregarding the differences between unrestricted free adaption and guided configuration. Every SME approach represents an individual tradeoff between the effort to design a method and the flexibility in terms of possible choices during the method design process [HB94]. On the one side there exist more rigid approaches to create a project-specific method. For example, configurable methods like V-Modell XT [Vm12] offer several variability points to adjust the method to the situation. On the other side, less rigid SME approaches are more flexible, in terms of the variety and specialization of the creatable methods, however, the design of methods requires more effort and more expertise.

A particular group of these less rigid SME approaches is called assembly-based SME [BS98]. The basic idea is to maintain a repository of predefined method building blocks, e.g., called method fragments [Br96], method chunks [Ro96], method components [GL98], or as in our case method services [Ro09]. Based on the situational context, method building blocks suitable for the current project are selected and assembled to a method (see **Figure 1**). The flexibility of this approach is restricted only by the set of available building blocks. These can be defined on-the-fly without requiring changes to existing method building blocks. Hence, it is possible to incorporate, e.g. the latest best practices.

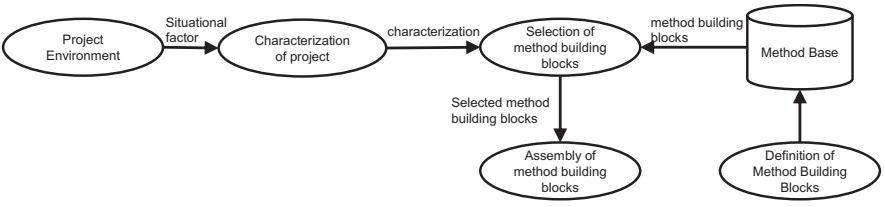


Figure 1. Assembly-based SME (cf. [Br96])

The major drawback of assembly-based SME is that creating meaningful methods requires a certain level of method engineering knowledge as it is more tedious and error-prone than configuration-based SME, where the possible configurations are already known beforehand. In this we see the main reason why assembly-based SME has not achieved noticeable attention in industry. Suitable building blocks have to be identified and combined in such a way that a consistent method is created. Furthermore, the method has to comply with the requirements imposed by the situation. To benefit from the

advantages of assembly-based SME, we introduce the new concept of method patterns to support the method engineer following that approach in his work.

Method patterns represent methodological aspects and quality constraints that shall be incorporated into the method, e.g., "iterative development" or "use of quality gates". They are combined to form a "method frame", which ensures that the combined method building blocks do not violate the pattern-specific properties. For example, a method pattern that prescribes the creation of a specification can be combined with a method pattern for iterative development. That ensures that in the created method that specification is created iteratively.

This paper is organized as follows. In Section 2 we use ePassport system development as an example to illustrate the rationale for assembly-based SME with method patterns. In Section 3 we exemplify the use of method patterns by combining patterns of a plan-driven software engineering method with patterns reflecting agile aspects. We conclude with a discussion of our contributions and the planned future work in Section 4.

2. Motivational Scenario

We use the following scenario as a running example to illustrate why configuration-based SME may fall short and to motivate assembly-based SME and the benefits of method patterns. It is based on real life industry projects carried out by one of our co-operation partners. The project in the scenario deals with the introduction of a distributed ePassport system connected to several national (e.g. border control, civil register) and international (e.g. Interpol) databases and information systems. Typically, such a system is not developed by the government organization itself, but by a supplier that is awarded the project after a public tender of the government organization (acquirer). Normally, a passport domain expert is the project manager. Right before the project's start the expert chooses the software development method of the project. As domain experts usually have no particular SME knowledge they choose fixed off-the-shelf methods or create a method using configuration-based SME, e.g., V-Modell XT [Vm12].

Figure 2 illustrates the lifecycle of such a project with the decision gates of V-Modell XT-based methods.[1] Each decision gate, depicted by a left leaning parallelogram, marks the end of a lifecycle phase, where the produced deliverables are examined.

Different from other domains, V-Modell XT has not been established as a standard for ePassport system projects. Nevertheless, the scope and formality of the methods created with V-Modell XT define a frame for the legal and commercial cooperation of acquirer and supplier. The desirable characteristics include:

- the work division between acquirer and supplier, e.g., support of tender activities
- the definition of formal documents, e.g. for legal and commercial reasons

[1] As we use V-Modell XT as an illustrative example, we abstract from different project execution strategies.

- the definition of formal handover activities, e.g., for legal and commercial reasons
- coverage of the lifecycle presented in **Figure 2**

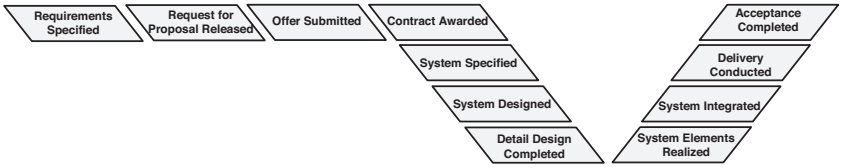


Figure 2. Sequence of decision gates in V-Modell XT-based methods

According to experience, the document-centric philosophy of methods created with V-Modell XT and their rigid formality are however also seen as the root for severe problems in practice. Typically, stakeholder are not familiar with ePassport technology and do not understand the implications caused by "just an additional chip on the passport". Therefore, they have difficulties to formulate all their requirements upfront and the requirements specification does not reflect the stakeholders' real intends. However, by the document-centric nature of a V-Modell XT-based method, the requirements specification is the main and dominant source of information for the supplier during the specification and development. Only little participation of the acquirer takes place during development and the stakeholders often do not see the system before it is ready to be delivered. Flaws uncovered then have to be removed at high costs.

In order to improve the situation, the method to use for the ePassport project shall therefore incorporate aspects of the agile software development philosophy [Be12] that fosters information exchange and collaboration. However, V-Modell XT is not designed to create a method that exhibits the following characteristics:

- iterative and incremental development towards decision gates
- informal coordination meetings
- sharing of intermediate work results

Using the scenario as an example we illustrated the limitations of configuration-based SME. Assembly-based SME, in contrast, allows incorporating method building blocks of different methods, especially following different development philosophies, as requested in our example. However, the available literature on assembly-based approaches provides no formal guidance for people without any particular SME knowledge during the method construction (e.g., [GH98, Fi09]). Additionally, if the effort to create a method is too high, the project manager will not be able to timely create a method for the initiated project. We therefore propose the concept of method patterns, which are used additionally to method building blocks during the method construction. They encode methodological aspects and quality constraints like the required presence and order of activities. In our Scenario the project manager could use and combine these patterns to assure that the method creates all the documents required by V-Modell XT and additionally shows the desired agile characteristics. Violated constraints of method patterns provide him with additional guidance during the method construction.

3. Situational Method Engineering with Method Patterns

In this section we exemplify the use of method patterns by the assembly-based creation of a method that incorporates the desired characteristics described in Section 2. On the one hand we define two *method patterns* that embody the essential constraints of these methods. For V-Modell XT we create a method pattern that reflects the development process of V-Model XT with the order of its decision gates. For Scrum we create a method pattern for the sprint loop. On the other hand we define *method services*, method building blocks that reflect the software development activities and artifacts of these methods. Recall that method patterns constrain the assembly of method building blocks (method services). By combining the method patterns and respective method services we could reconstruct the two original methods. However, we show how the combination of method patterns from both methods guides the method engineer to create a hybrid method, which maintains the order of document creation conforming to V-Modell XT, but ensures that they are developed in sprint loops.

3.1 Extraction of Method Patterns and Method Services from V-Modell XT

As stated in Section 2 methods created with V-Modell XT define a flow of decision gates that have to be passed to accomplish the project (see **Figure 2**). In V-Modell XT at each decision gate a set of documents has to be approved using the activity "Project coming to a progress decision". For example, **Figure 3** shows on the right the three documents that have to be approved for the decision gate "System Specified" depicted on the left. Consequently, these documents have to be produced before the decision gate can be passed. Thus, in methods based on V-Model XT the sequence of decision gates indirectly specifies the order of activities that have to be performed. For example, for the decision gate "System Specified*"* the document "Overall System Specification" has to be created with a software development activity called "Preparing Overall System Specification" and it has to be approved like every other document using the activity "Project coming to a progress decision".

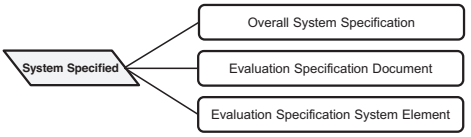


Figure 3. Decision gate "System Specified" and related documents of V-Modell XT

We now translate the flow of decision gates into a method pattern, by first creating a method pattern for every decision gate and then combining them into an overall method pattern that reflects the V-Modell XT development process. We later reuse this method pattern when creating the hybrid method.

Figure 4 illustrates the relationship between the constituents of a method pattern and method services. A method pattern consists of *method compartments*, denoted by dotted rectangles. These are restricted by the attached *pattern constraints*, depicted by grey

boxes. Pattern constrains restrict their respective method compartment, because the hosted method services must fulfill these constraints.

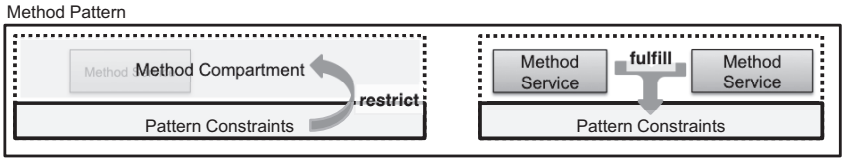


Figure 4. Overview of the relationship between method patterns, compartment, and services and pattern constraints

Figure 5 shows a method pattern that encodes the concrete decision gate "System Specified*"*. The method pattern consists of two consecutive method compartments. The first method compartment fulfills its pattern constraints, if it hosts a method service that has the respective artifact among its outputs, for each of the three artifacts named in **Figure 3**. The second method compartment has to contain a method service that has the value `reviewing` assigned to its attribute `activity type`. Thus, the method pattern describes, that method services have to create the three named documents and that they have to be followed by a method service encapsulating a reviewing activity. **Figure 6** shows a combination of method services that fulfills the constraints.

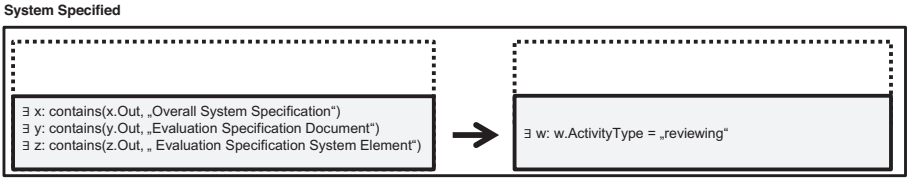


Figure 5. Method pattern for the decision gate "System Specified"

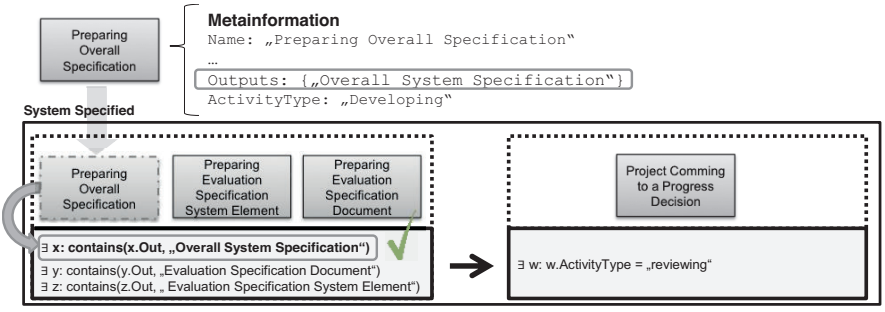


Figure 6. A method pattern with fulfilled pattern constraints

Each software development activity and its input and output documents are translated to a method service with respective input and output artifacts. For example, the method service "Preparing Overall Specification" encapsulates the equally named V-Modell XT activity and its output "Overall System Specification" (see **Figure 6**). Additional metain-

formation specifies that this is a development activity: the attribute `ActivityType` of the method service has the value `developing` assigned to it. Thus the first method service in the first method compartment "Preparing Overall Specification" fulfills the first pattern constraint as it has the required artifact among its outputs. Similar, the other three method services fulfill the remaining pattern constraints. **Figure 6** is only an example for a fulfilled method pattern; we do not combine method patterns and method services yet.

To obtain a method pattern that reflects the development process of V-Model XT with the order of its decision gates we chain the concrete method patterns of all decision gates to an overall "V-Modell XT" method pattern. **Figure 7** illustrates this for the three consecutive decision gates depicted on the left. The three method patterns are combined to a new method pattern that has no pattern constraints on its own, but specifies the order of the contained method patterns "System Specified", "System Designed" and "Detail Design Completed".

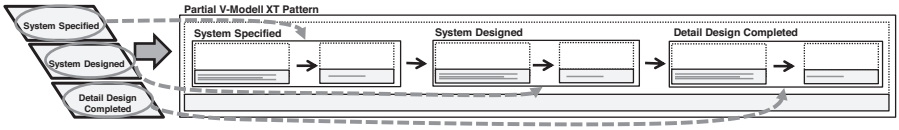


Figure 7. Decision gate sequence reflected as a method pattern

3.2 Extraction of Method Patterns and Method Services from Scrum

Scrum is a widespread agile development method that we use in our example to define agile method patterns and agile method services. One of the core aspects of Scrum is a time-boxed execution loop called "Sprint" that is repeated throughout the duration of the project. **Figure 8** shows the method pattern "Sprint Loop", which requires method services that reflect agile activities. The method pattern consists of the three sub method patterns "Sprint Planning", "Agile Construction" and "Sprint Review" that are combined to a loop. For example, "Agile Construction" describes that all the method services in the respective method compartment have to either encapsulate developing activities or contain the backlog artifact among their inputs. Additionally, the use of a method service named "Standup Meeting" is prescribed and has to be present in the method compartment.

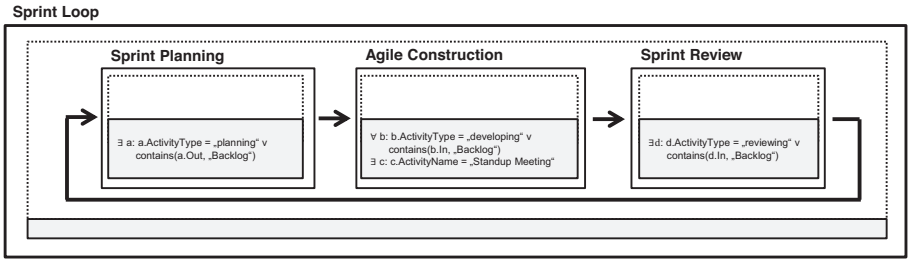


Figure 8. The Sprint Loop pattern extracted from Scrum

Based on the Scrum guide [SS11] the method services "Sprint Planning", "Standup Meeting", "Update Backlog" and "Sprint Review*"* are identified. They fit into the appropriate method compartments of the method pattern depicted in **Figure 8**.

3.3 Creation of a Situational Method for ePassport System Development

For a typical method creation procedure, the previously constructed and presented method patterns and method services would have been identified and retrieved from the method base instead of being defined from scratch (see **Figure 1**). The next step now is their combination. Different from traditional assembly-based approaches we first combine method patterns and then place method services into the method compartments of these patterns. In our example the Project Manager picks the overall "V-Modell XT" method pattern, to assure the conformance to the prescribed order of activities. In addition he adds a "Sprint Loop" method pattern into every decision gate method pattern, because he wants it to be executed in an agile manner. **Figure 9** illustrates this for the decision gate "System Specified". The combination of method patterns now prescribes and ensures that the created method will contain a "Sprint Loop" in every decision gate method pattern and that all necessary activities of V-Modell XT are executed in the right order. Compared to other assembly-based approaches, with this frame of method patterns it is much easier to decide, which method services to use and where in the process to put them. **Figure 10** shows the combination of method patterns after adding method services to fulfill the pattern constraints depicted in **Figure 9**.

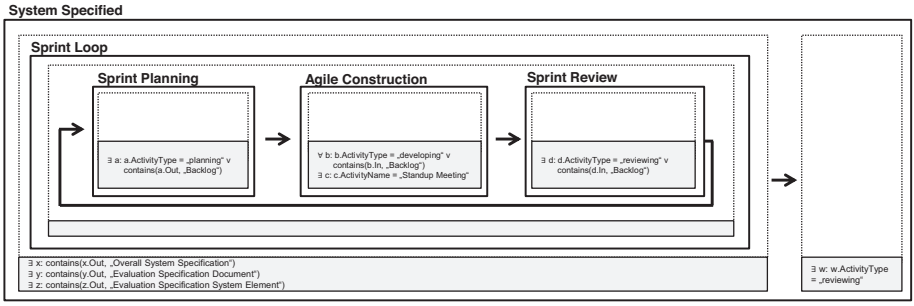


Figure 9. Combined method patterns derived from V-Modell XT and Scrum

With the method creation state shown in **Figure 10** there could be additional refinement iterations. For example, in the method compartment of "Agile Construction" illustrated in **Figure 11** the "Sprint Loop" method pattern could be used again to model the daily Scrum, which is a daily sprint loop, of the Scrum method. As this additional formality is not desired for this ePassport project, the method creation procedure is finished by (manually) connecting the method services with control flow. According to the practices in the Scrum Guide the control flow specifies that the work is carried out in a loop, where "Standup Meeting" precedes the parallel execution of the development method services. "Update Backlog" is executed continuously in parallel.

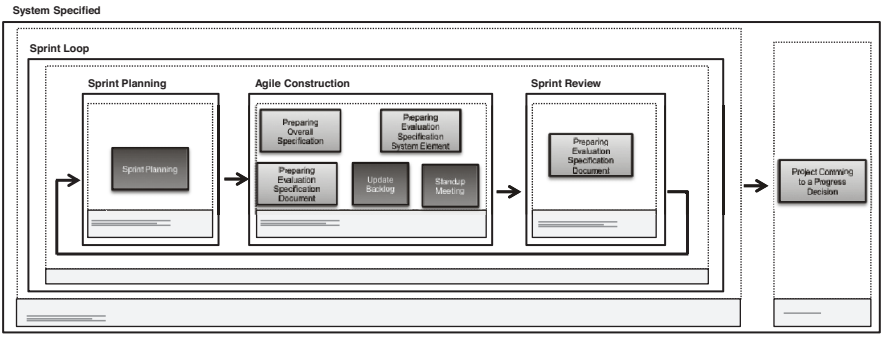


Figure 10. Combination of method services that fulfill the constraints of the method patterns in Figure 8

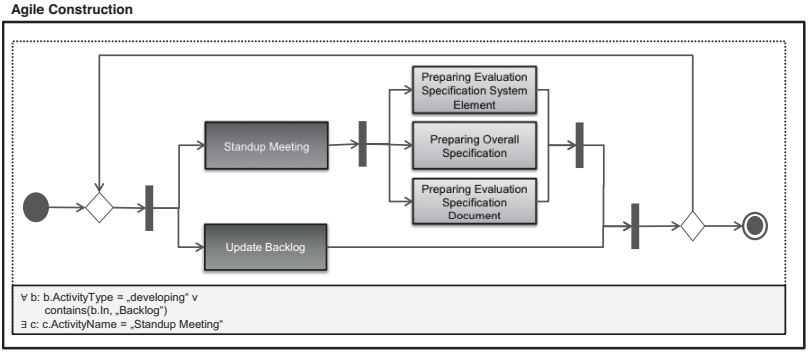


Figure 11. Method services of a method compartment connected with control flow

4 Conclusions and Future Work

Using a realistic example scenario from the ePassport system development domain we exemplify that in some cases more rigid SME approaches are not flexible enough to reflect the situational context, e.g., agile aspects in our example. Here, assembly-based SME provides the required flexibility, however, we criticize that assembly-based approaches require too much method engineering knowledge and offer insufficient support to create methods of good quality. We exemplified the use of method patterns and show how they can guide in choosing and combining suitable method building blocks, supporting the method engineer in his work. Doing so we also show how building blocks from different methods can be incorporated. The resulting created method preserves the flow of activities and the document-based approvals at specific decision gates known from V-Modell XT. In addition, by incorporating sprints of Scrum, there are fixed cycles, where results are planned, produced, presented, and discussed between the approvals of two consecutive decision gates.

Although we argue, that by the use of method patterns we have an advantage over pure assembly-based approaches, the additional freedom compared to more rigid approaches as configuration-based SME still requires more time and more skills to create the method. However, we will be able to gradually improve on the status quo in the future. We continue our work in two directions. In this paper we focused on activities and control flow. First, we work on formalizing other aspects of methods like roles, artifact lifecycles and object flow. Second, applying this approach in practice is feasible only with sufficient tool support. In a project with an industrial partner we work on an expert system that supports the different activities of method creation. We are also evaluating how such method specifications can be enacted in terms of a workflow engine, task management and integrated tooling like version control and plan to evaluate the whole approach in their industry projects.

References

[Be12] Beck, K. et al.: Manifesto for Agile Software Development, http://agilemanifesto.org/

[Bo03] Boehm, B.W., Turner, R.: Observations on Balancing Discipline and Agility. In: ADC 2003, pp. 32–39. IEEE Computer Society, Los Alamitos, Calif (2003)

[Br96] Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Inf. Softw. Technol. 38, 275–280 (1996)

[BS98] Brinkkemper, S., Saeki, M., Harmsen, A.F.: Assembly Techniques for Method Engineering. In (Pernici, B., Thanos, C. eds.): CAiSE '98, pp. 381–400. Springer, Berlin (1998)

[Cm10] CMMI Product Team: CMMI for Development, Version 1.3. Improving processes for developing better products and services Pittsburgh, Pennsylvania (2010)

[ES10] Engels, G., Sauer, S.: A Meta-Method for Defining Software Engineering Methods. In (Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. eds.): Graph Transformations and Model-Driven Engineering, pp. 411–440. Springer, Berlin (2010)

[Fi09] Firesmith, D.G.: The method framework for engineering system architectures. CRC Press, Boca Raton (2009)

[GH98] Graham, I., Henderson-Sellers, B., Younessi, H.: The OPEN process specification. ACM Press, New York (1997)

[GL98] Goldkuhl, G., Lind, M., Seigerroth, U.: Method Integration: The Need For A Learning Perspective. IEE Proceedings Software 145, 113–118 (1998)

[HB94] Harmsen, F., Brinkkemper, S., J. L. Han Oei: Situational method engineering for informational system projects. In (Verrijn-Stuart, A.A., Olle, T.W. eds.): CRIS'94, pp. 169–194. North-Holland Publishers, Amsterdam (1994)

[HR10] Henderson-Sellers, B., Ralyté, J.: Situational Method Engineering: State-of-the-Art Review. j-jucs 16, 424–478 (2010)

[Kr99] Kruchten, P.: The rational unified process. An introduction. Addison-Wesley, Reading, Mass (1999)

[Ro96] Rolland, C., Prakash, N.: A proposal for context-specific method engineering. In (Brinkkemper, S., Lyytinen, K., Welke, R.J. eds.): Method Engineering: Principles of method construction and tool support, pp. 191–208. Chapman & Hall, London (1996)

[Ro09] Rolland, C.: Method engineering: towards methods as services. Softw. Process: Improve. Pract 14, 143–164 (2009)

[SS11] Schwaber, K., Sutherland, J.: The Scrum Guide (2011)

[Vm12] V-Modell XT (english version), http://v-modell.iabg.de/v-modell-xt-html-english/index.html

Traceability – Nutzung und Nutzen

Data Lineage goes Traceability - oder was Requirements Engineering von Business Intelligence lernen kann

Andreas Ditze

MID GmbH
Kressengartenstraße 10
90402 Nürnberg
a.ditze@mid.de

Abstract: Data Lineage ist als Konzept in Business Intelligence Systemen unabdingbar, um z.B. die Verwendung bzw. Verfolgung von Attributen über die verschiedenen Ebenen einer Data Warehouse Infrastruktur zu ermöglichen.

Ausgehend von den Attributen des fachlichen Datenmodells über die Spalten des physischen Datenbankmodells der operativen Systeme werden die Informationen über die Data Warehouse Schicht bis zur Data Mart Schicht per Data Lineage verfolgt.

Diese Konzepte, Mechanismen und entsprechende Werkzeugunterstützung lassen sich nahezu 1:1 auf die Traceability-Anforderungen eines Requirements Engineering & Management anwenden.

Der Beitrag zeigt anhand von kleinen Praxisbeispielen die Vergleichbarkeit der beiden Konzepte auf und stellt eine entsprechende Werkzeugunterstützung vor.

1 Einleitung

Traceability ist extrem hilfreich, aber auch aufwendig aufzubauen und damit teuer in der Erstellung und im Unterhalt. Deshalb müssen effektive und effiziente Methoden und Werkzeuge zum Aufbau und Pflege der für eine Traceability notwendigen Abhängigkeiten zwischen den Ergebnisartefakten aufgebaut und genutzt werden. Dabei ist es besonders wichtig, dass die Abhängigkeiten möglichst automatisch während des Entwicklungsprozesses erstellt bzw. abgeleitet und gepflegt werden. Dieser Beitrag wird anhand einer modellbasierten Vorgehensweise und einer Anlehnung an das Konzept des Data Lineage aus dem Business Intelligence Umfeld aufzeigen, wie in der Praxis und unter Verwendung eines entsprechenden Modellierungswerkzeugs Traceability genutzt wird.

Data Lineage (oder auch Datenherkunft) ist sinngemäß die Fragestellung, zu gegebenen (meist) aggregierten Datensätzen die ursprünglichen Datensätze zu bestimmen, aus denen sie entstanden sind.

Für Ergebnisartefakte in Softwareentwicklungsprozessen gilt ähnliches. Auch hier möchte man wissen, aufgrund welcher Anforderung eine bestimmte Softwarekomponente oder –funktion entstanden ist, bzw. welcher Testfall sich auf welche Anforderung von welchem Stakeholder bezieht. Warum also nicht die gleichen Mechanismen und Techniken dafür einsetzen?

2 Data Lineage als Grundprinzip

Zur Verfolgung der Datensätze über die verschiedenen Ebenen einer Data-Warehouse-Architektur werden die jeweiligen Attribute auf den unterschiedlichen Ebenen mit Abhängigkeiten verbunden. Zusätzlich kann die Abhängigkeit noch mit einer Transformationsregel erweitert werden. Mit einem einfachen Abhängigkeitsgraphen kann so die Datenherkunft jedes einzelnen Attributs eines Datensatzes bestimmt werden.

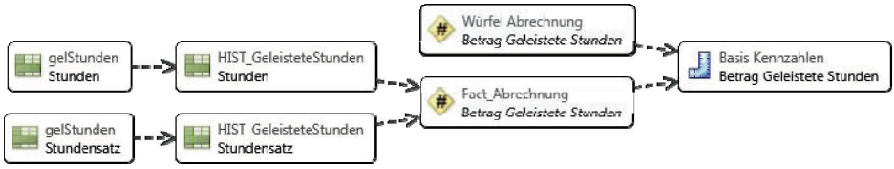


Abbildung 1: Data Lineage für die Basis-Kennzahl "Betrag Geleistete Stunden"

Die Abbildung 1 zeigt die Verfolgung der Zusammensetzung und Herkunft einer Basis-Kennzahl „Betrag Geleistete Stunden“. Ausgehend von den links dargestellten Attributen „Stunden“ und „Stundenstatz“ aus der Ladetabelle „gelStunden“ über die historisierten Tabellen der Data-Warehouse-Schicht bis zur Faktentabelle und der Kennzahl aus der Data-Mart-Schicht.

Dieses Grundprinzip lässt sich auch in der Softwareentwicklung anwenden, nur das hier die Art der Elemente vielfältiger ist. Es handelt sich nicht nur um Attribut von Entitäten, Spalten von Tabellen oder Würfeln, sondern um unterschiedlichste Instanzen der verschiedenen Metamodelle (u.a. UML 2, BPMN 2). Meist sogar noch ergänzt um textuelle Artefakte wie funktionale Anforderungen, Abnahmekriterien, Risiken und Testfälle. Dadurch ergeben sich eine Reihe von möglichen Beziehungen und Abhängigkeiten, die entweder von Hand im Rahmen der Modellierung explizit erzeugt oder implizit durch die Metamodell-Beziehungen hergestellt werden.

Zwei Beispiele für implizite Metamodell-Beziehungen sind die „Owned Element“ Beziehung (Ownership) eines UML-Attributes zur UML-Klasse oder einem BPMN-Task zum BPMN-Prozess. Diese Beziehungen werden nicht durch Abhängigkeiten modelliert sondern werden im Rahmen der Modellbildung erstellt.

Beispiele für explizite Abhängigkeiten sind die manuelle Zuordnung von Anforderungen zu Anwendungsfällen, UML-Klassen zu BPMN-Datenobjekten oder Rollen bzw. Organisationseinheiten zu BPMN-Lanes bzw. BPMN-Pools.

Beispielhaft soll dies an einem Übersichtsschaubild (sog. Whiteboard-Diagramm) für ein kleines Modellierungsprojekt gezeigt werden:

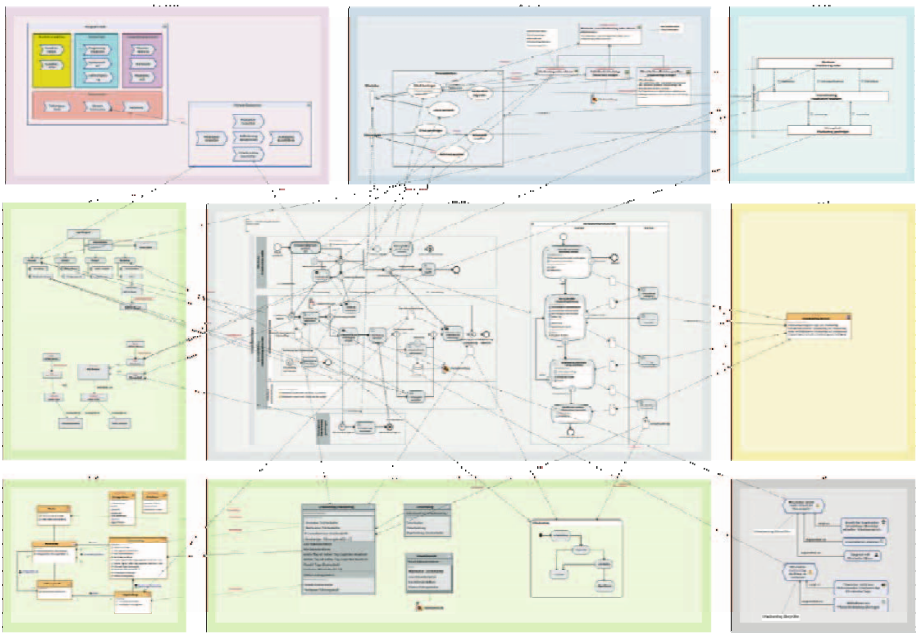


Abbildung 2: Übersichtsschaubild mit allen Abhängigkeiten zwischen den Modellelementen

Die Abbildung 2 zeigt die unterschiedlichen Modellierungsartefakte und die verschiedene Beziehungen (explizit oder implizit erstellt) untereinander. Jede Beziehung (Abhängigkeit) kann in einer Traceability-Analyse zur Verfolgung über die verschiedenen Modellierungsebenen genutzt werden.

So entsteht z.B. ein Traceability-Pfad von einer funktionalen Anforderung → UML-Anwendungsfall → BPMN-Kollaboration → BPMN-Prozess → BPMN-Task → BPMN-Datenassoziation → BPMN-Datenobjekt → Geschäftsobjekt → UML-Klasse → ER-Entität → DB-Tabelle → Datenbank.

Über diesen Pfad kann direkt die Frage beantwortet werden, welche Datenbank von der Änderung einer Anforderung betroffen ist (in Pfeilrichtung) bzw. welcher Geschäftsprozess vom Ausfall einer Datenbank betroffen ist und welcher Stakeholder informiert werden muss (entgegen der Pfeilrichtung).

Neben der Darstellung in Abbildung 2 mit vollständigen Diagrammen in der jeweiligen Modellierungsnotation ist auch eine Darstellung als einfacher Abhängigkeitsgraph

möglich. Dieser Graph stellt nur die einzelnen Modellelemente (z.B. Anforderung, Geschäftsobjekt, Klassenattribut, Entitätsattribut) mit ihren Abhängigkeiten dar und ermöglich damit eine schnelle Verfolgung der Auswirkung einer Änderung.

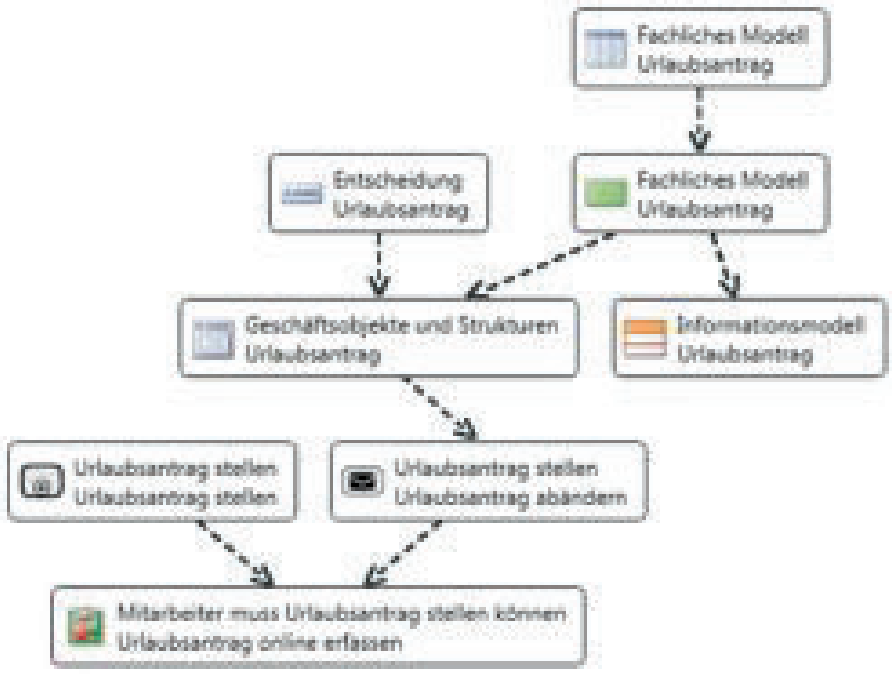


Abbildung 3: Dynamischer Abhängigkeitsgraph für beliebige Modellelemente

Mit Hilfe der in Abbildung 3 gezeigten Darstellungsmöglichkeit wird Traceability sowohl beim Erstellen der Abhängigkeiten als auch bei der Pflege und Auswertung beherrschbar. Viele der Abhängigkeiten entstehen implizit beim Modellieren oder durch einfache Drag-&-Drop-Aktionen bzw. automatisierte Modell-zu-Modell-Transformationen.

Die nachfolgende Abbildung 4 zeigt eine spezielle Applikation („Beamer“), die zur interaktiven Erstellung und Verwaltung von Abhängigkeiten zwischen unterschiedlichen Quellen und Zielen geeignet ist. Dabei können verschiedene sog. Abbildungsthemen definiert werden, bei denen die Auswahl von Quell- und Zielelementen und der dazwischen anzulegenden Abhängigkeit vordefiniert wird.

Ein beispielhaftes Abbildungsthema ist das Mapping von Attributen der Entitäten des konzeptionellen Datenmodells auf Spalten von Tabellen eines physischen Datenbankmodells. Dabei werden als Quelle die Attribute und Entitäten des konzeptionellen Datenmodells angezeigt und als mögliche Ziele die bereits vorhandenen Spalten und Tabellen des physischen Datenbankmodell. Nun kann entweder eine Verbindung (Abhängigkeit) zwischen bereits vorhandenen Modellelementen hergestellt

werden oder man erzeugt aus einem Quellelement ein neues Zielelement und verbindet beide gleichzeitig mit einer entsprechenden Abhängigkeit.

Vorhandene Abhängigkeiten werden durch eine grüne Verbindungslinie zwischen dem Quell- und Zielelement im oberen Bereich der Applikation angezeigt und zusätzlich in der Bearbeitungsliste (unterer Bereich) aufgelistet.

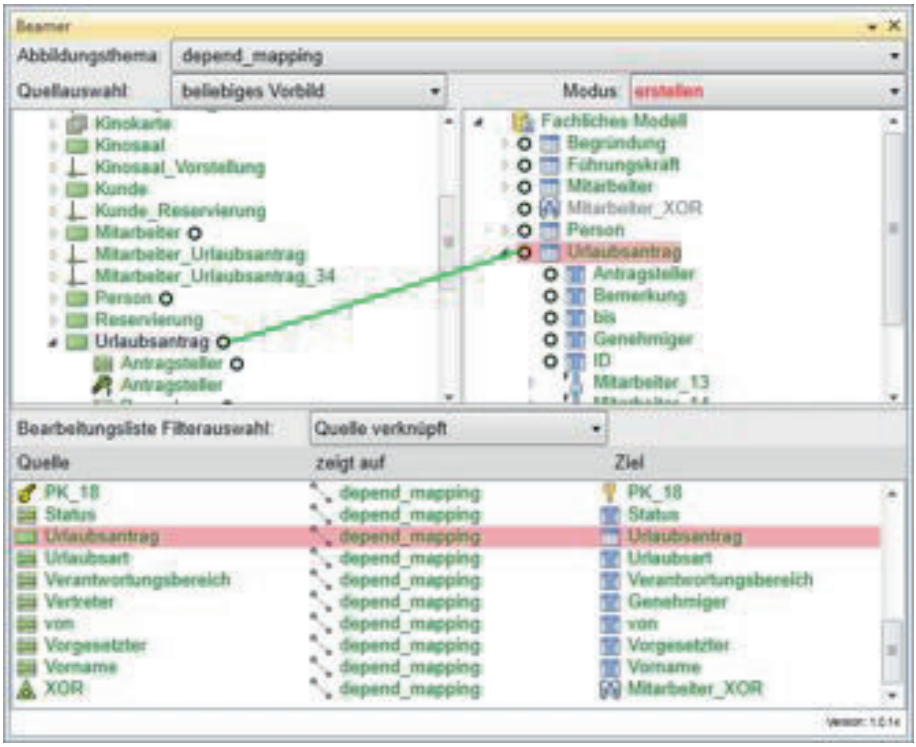


Abbildung 4: Der "Beamer" - eine interaktive Applikation für Abhängigkeiten

In der Bearbeitungsliste der „Beamer“-Applikation (unterer Bereich in Abbildung 4) wird durch eine entsprechende Einfärbung der Einträge deren Aktualität gekennzeichnet. So werden Einträge, bei denen das Modellelement der Quelle ein jüngeres Änderungsdatum hat als das Ziel-Modellelement rot eingefärbt. Dadurch können auch bei einer großen Anzahl an vorhandenen Abhängigkeiten die notwendigen Anpassungen nach einer Änderung des Quellmodells schnell ermittelt werden.

Hierzu kann die Bearbeitungsliste, wie in Abbildung 5 dargestellt, auch entsprechend gefiltert werden.

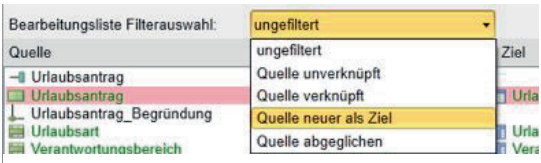


Abbildung 5: Filterauswahl der Bearbeitungsliste

Mit den im Beitrag gezeigten Beispielen sollen die verschiedenen Unterstützungsmöglichkeiten eines modernen Modellierungswerkzeugs beim Aufbau und Pflege der Abhängigkeiten zwischen Modellelementen vorgestellt werden.

Der Anwender erstellt im Rahmen des Modellierungsprozess durch verschiedenste manuelle, teilautomatische oder vollautomatische Verfahren entweder implizit oder explizit die unterschiedlichen Abhängigkeitsbeziehungen zwischen den Quell- und Zielmodellelemente. Diese Abhängigkeiten können mit den dargestellten Funktionalitäten ausgewertet und gepflegt und damit nutzbar gemacht werden.

3 Fazit

Traceability ist in einer modellbasierten Vorgehensweise beim Einsatz von modernen Modellierungswerkzeugen mit unterschiedlichen Darstellungs- und Auswertungsfunktionalitäten effizient und effektiv einsetzbar.

Quelle für alle Abbildungen: Modellierungsplattform Innovator®

A Model Management Framework for Maintaining Traceability Links

Thomas Beyhl, Regina Hebig and Holger Giese
Hasso Plattner Institute for IT Systems Engineering at the University of Potsdam
Prof.-Dr.-Helmert-Street 2-3, 14482 Potsdam, Germany
{prename}.{surname}@hpi.uni-potsdam.de

Abstract: In MDE diverse modeling and model transformation languages are applied to describe and derive the envisioned system. Traceability is a prerequisite for maintaining consistency between different development artifacts. Thereby, the usefulness of traceability links increases with their completeness and correctness. In practice, automatic creation and maintenance of traceability links is required to be useful. This is addressed by heuristic approaches that derive traceability information statically or by model transformation technologies that provide traceability links as additional execution result. However, the maintenance of traceability links for a set of diverse languages and transformation technologies as combined in MDE is still a challenging task. In this paper, we present a framework that provides and treats all traceability information using the common format of hierarchical megamodels. Thereby, different approaches for gaining traceability information can be combined. Information provided by transformation technologies is translated into this common format.

1 Introduction

In MDE diverse modeling and model transformation languages are applied to describe and derive an envisioned system. Traceability is a prerequisite for maintaining consistency between different development artifacts. Thereby, the usefulness of traceability links increases with their completeness and correctness. In practice, a huge amount of development artifacts is used to describe the envisioned system. Therefore, the manual creation and maintenance of traceability links is not feasible and automatic creation and maintenance of traceability links is required. This is addressed by heuristic approaches that derive traceability information statically, e.g., [An02], or by model transformation technologies, which provide traceability links as additional execution result, e.g. [Jo05]. Winkler et. al [WP10] give a survey of traceability in requirements engineering and MDE. However, the maintenance of traceability links for a set of diverse languages and transformation technologies as combined in MDE is still a challenging task. First, many transformations are performed manually or by transformation technologies, which provide no traceability information. Second, the proprietary format of traceability links provided by different transformation technologies prevents further treatment of this information for analysis of the whole set of artifacts, e.g. for impact analysis or consistency checks. In the following, we present a framework for maintaining traceability links that addresses the diversity of modeling and model transformation languages in MDE. We developed the framework as

a basis of a set of research projects that build on traceability, e.g. a model transformation composition framework [Se11], a framework extension for version control capabilities[1] and a corresponding model management build server [SHG12]. Thereby, we address the two challenges by allowing a combination of different traceability approaches for gaining traceability information and by providing and treating all traceability information using the common format of hierarchical megamodels.

2 Model Management Framework

Barbero et. al [BB08] introduce the concept of megamodels. Such megamodels capture models and relationships between models. Thus, megamodels are well suited for capturing, providing and maintaining diverse models and traceability links between them. Hierarchical megamodels combine high-level traceability models (megamodels) and low-level traceability models (models containing fine-grained traceability links) [SNG10]. Thereby, hierarchical dependencies between high-level and low-level modeling artifacts and between traceability links are defined. Thus, traceability information is combined into one common traceability model. In our case, a hierarchical megamodel provides a logical view of the local workspace by capturing representatives for artifacts (e.g. models, model elements, source code). The framework[2] presented here a) supports an easily extensible set of modeling and model transformation languages, b) monitors artifact changes and updates the set of representatives within the hierarchical megamodel concerning the development artifacts in the local workspace, and c) gains and maintains traceability links automatically, based on monitored development artifact changes, model transformation executions and available traceability approaches.

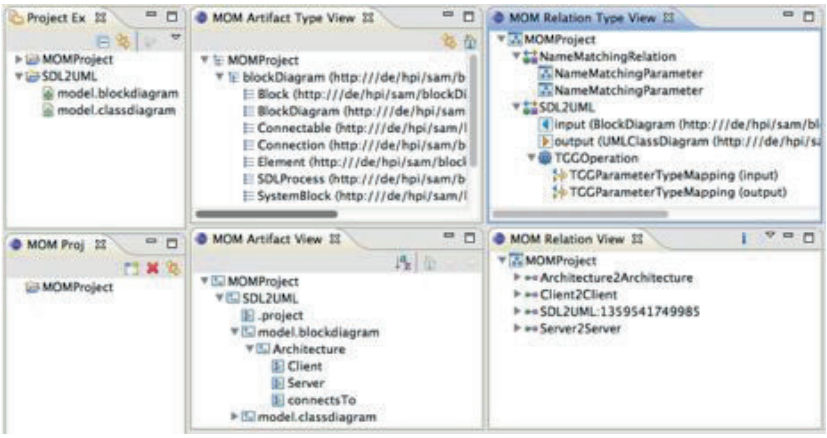


Figure 1: Views of the hierarchical megamodel

[1] http://www.hpi.uni-potsdam.de/giese/gforge/mdelab/?page_id=108
[2] http://www.hpi.uni-potsdam.de/giese/gforge/mdelab/?page_id=63

2.1 Type System

Information about physical artifacts, types, model operations and traceability links are captured in a hierarchical megamodel. Development artifacts (e.g. models, model elements, code) in the local workspace are represented as *artifacts* within the hierarchical megamodel. Correspondingly, all captured and retrieved traceability information is represented as *relations*. The framework includes an additional type layer, where metamodels for used *artifact types* can be registered to improve the quality of the information in the hierarchical megamodel. This type information is used to assign an artifact type to each captured artifact. Thus, the framework can support arbitrary modeling languages. In addition, *relation types* can be specified to describe the signature of model operations and traceability links (i.e. information about types of input and output artifacts of a relation). To support an extensible set of model transformation languages a plug-in mechanism is provided. Thereby, adapters can be plugged-in, which hide the model transformation technology internals. For example, we provide adapters to execute ATL[3] and Xpand[4] model transformations. Adapters are used to trigger the execution of model transformations and to retrieve traceability information about the execution. When executing a model transformation an *executable relation* is created within the hierarchical megamodel and the corresponding relation type is assigned. Traceability links are represented by *non-executable relations* in the hierarchical megamodel, which connect input and output artifacts. Figure 1 depicts different views (artifact types, relation types, artifacts, relations) of the hierarchical megamodel. For example, the application of the model transformation $SDL2UML$ and automatically created traceability links (e.g. $Client2Client$) are depicted.

2.2 Updating Artifact Representations

Development artifacts can be registered for being monitored by the framework. Changes on these registered development artifacts are monitored and a notification mechanism provides change events when these monitored development artifacts change. Further, the model transformation adapters may throw change events when model transformations lead to a change, deletion or creation of models or model elements. Events are propagated to the workspace builder, which is responsible to update the internal hierarchical megamodel (i.e., create, update, or delete artifacts). Further, events are propagated to tools that base on the framework.

2.3 Maintaining Traceability Information

Special kinds of the mentioned tools are traceability adapters, which enrich the hierarchical megamodel with statically derived traceability information implementing heuristic

[3] http://www.eclipse.org/atl/

[4] http://www.eclipse.org/modeling/m2t/?project=xpand

techniques. Furthermore, model transformations are directly triggered via our framework using model transformation adapters. Thereby, the execution of model transformations can be monitored in more detail. For example, also proprietary forms of traceability links that are created during the execution of model transformations (at runtime), e.g. [Jo05], can be translated to be stored in our hierarchical megamodel. We implemented different technology-specific model transformation adapters. Thereby, we experienced how strong the quality of retrievable traceability information and also the control over execution configuration parameters depends on the model transformation technology. For example, it turned out that the very flexible model transformation technology Jet[5] is hard to control and provides only little traceability information. For example, the question which target artifacts are generated cannot be answered before executing the model transformation. This circumstance makes the integration of Jet into model transformation chains unfeasible. In contrast, the less flexible and more restricted model transformation technology Xpand, allows better control and traceability information can easily be retrieved during the execution of the model transformation. For all gained traceability information traceability links (non-executable relations) are created within the hierarchical megamodel. Traceability links may become incorrect when a model transformation was executed or artifacts were changed manually. Based on the automated update of the hierarchical megamodel the framework recognizes such situations and identifies the need to update or delete specific traceability links as well. Traceability links retrieved by monitoring model transformation executions are deleted (or marked as invalid), since they can only be restored when executing the model transformation again. However, traceability adapters can directly update all traceability links, which they had created.

3 Related Work

The AM3Core [BB08] provides a metamodel for megamodels that includes the concept of models, relationships between models and chains of relationships. The Model Management Tool Framework (MMTF) [Sa07] is an environment that enables different software developers to work on different related parts of models and their relationships with the help of a Model Interconnection Diagram (MID). However, both approaches are suitable for automatically maintaining and combining traceability links.

4 Conclusion

We presented a framework for capturing and maintaining artifacts and traceability links between them within a hierarchical megamodel. Thereby, we showed that different traceability approaches can be combined to establish a chain of traceability links within model transformation chains. In addition, we showed using the examples of ATL, Xpand, and Jet that the framework can be extended to support different technologies and that traceability

[5] http://www.eclipse.org/modeling/m2t/?project=jet#jet

information can be extracted automatically. The traceability information stored within the hierarchical megamodel and the events provided by the framework enable other tools to use these traceability information. The framework was successfully applied in different research projects. In future work, we focus on extending the set of model transformation adapters, e.g. QVT Operational[6].

Acknowledgement

The authors are grateful for the input of Andreas Seibel and the student assistants Henrik Steudel, Dmitry Zakharov, Arian Treffer, Johannes Dyk and Manuel Hegner.

References

[An02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[BB08] Mikaël Barbero and Jean Bézivin. Model driven management of complex systems: Implementing the macroscope's vision. *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 277–286, 2008.

[Jo05] Frederic Jouault. Loosely coupled traceability for ATL. In *Proceedings of European Conference on Model Driven Architecture workshop on traceability*, pages 29–37, 2005.

[Sa07] Rick Salay, Marsha Chechik, Steve Easterbrook, Zinovy Diskin, Pete McCormick, Shiva Nejati, Mehrdad Sabetzadeh, and Petcharat Viriyakattiyaporn. An Eclipse-based tool framework for software model management. *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 55–59, 2007.

[SHG12] Henrik Steudel, Regina Hebig, and Holger Giese. A Build Server for Model-Driven Engineering. In *6th International Workshop on Multi-Paradigm Modeling (MPM 2012)*. ACM, 2012.

[Se11] Andreas Seibel, Regina Hebig, Stefan Neumann, and Holger Giese. A Dedicated Language for Context Composition and Execution of True Black-Box Model Transformations. In *4th International Conference on Software Language Engineering (SLE 2011)*, pages 19–39, Braga, 2011.

[SNG10] Andreas Seibel, Stefan Neumann, and Holger Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software & Systems Modeling*, 9(4):493–528, September 2010.

[WP10] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software & Systems Modeling*, 9(4):529–565, September 2010.

[6] http://www.eclipse.org/projects/project.php?id=modeling.mmt.qvt-oml

UNICASE Trace Client: A CASE Tool Integrating Requirements Engineering, Project Management and Code Implementation

Alexander Delater, Barbara Paech

Institute of Computer Science, University of Heidelberg
Im Neuenheimer Feld 326, 69120 Heidelberg, Germany
{delater, paech}@informatik.uni-heidelberg.de

Abstract: Artifacts for requirements engineering, project management and code implementation are usually stored in separate tools, which makes traceability between these artifacts difficult. We developed the tool UNICASE Trace Client, which stores the aforementioned artifacts in a single environment with full traceability between all artifacts. In this paper, we describe the three traceability link creation process supported by our tool as well as its advanced features for traceability link usage.

1 Introduction

Requirements-to-code traceability reflects the knowledge where requirements are implemented in the code. The simple creation of such links is very important for a development project, as the manual creation can, for example, lead to higher development effort. In previous work [DNP12], we presented an approach that captures traceability links between requirements and code as the development progresses by using artifacts from project management called work items. Based on this approach, we developed the tool UNICASE Trace Client [UTC]. It is an extension to the model-based CASE tool UNICASE [UNI], which is an Eclipse plug-in developed in an open-source project. UTC integrates itself seamlessly in Eclipse and supporting plug-ins, e.g. Subversion, and supports three processes for traceability link creation between requirements, work items and code. Based on these links, it supports various features for traceability link usage.

The remainder of this paper is structured as follows: Section 2 describes the three traceability link creation processes and features supported by UTC. Section 3 concludes the paper and discusses future work.

2 UNICASE Trace Client

UTC incorporates artifacts from *requirements engineering* (features, functional requirements), *project management* (work items, sprints, developers) and *code* (code files, revi-

sions). A feature is realized in a sprint and is detailed in one or more functional requirements. Work items describe work to be done to realize functional requirements, they are assigned to developers, have a completion status and a due date. A work item must have one or more linked functional requirements and is contained in a sprint. A feature can be related to a work item, e.g. during bug fixing. One work item can create one or more revisions. A revision contains one or more changed code files and is stored in a version control system (VCS).

For using UTC, we presume the following situation in a development project and the used development process. First, a list of features and functional requirements exists. Second, a project manager has planned the implementation of the features in sprints and s/he has broken down the implementation schedule of the functional requirements into work items for the developers. Third, all work items are already assigned to developers.

2.1 Traceability Link Creation Processes

UTC uses work items to link requirements and code during development. As we presume that the implementation of the requirements is planned in work items, UTC captures links between the work item and the code that is created by its assigned developer. We identified three possibilities of developers to select a work item that is related to their implemented code. Developers can select a work item *before* they start the implementation of code (Process A), *during* implementation when they have created code but have not yet stored it as a new revision in a VCS (Process B), or *after* implementation when they have created code that is already stored as a revision in a VCS (Process C). These three processes are depicted in Figure 1. In general, developers should not perform any change in the code without a work item describing the realization of a requirement.

Requirements, work items and revisions are stored as artifacts in UTC. However, revisions only contain a subset of information (revision number, author, creation date and list of changed code files) that is stored in the VCS. More detailed information, e.g. what lines of code were changed in the revision, can be found in the VCS.

2.1.1 Process A) Select Work Item Before Implementation

First, the developer selects a work item from his/her list of assigned work items. While working on the work item and implementing new code or changing existing code, all requirements the developer looks at during implementation are automatically captured. For example, s/he may look at requirements to know what to implement. When finishing the implementation of the work item, the developer is asked to validate all captured requirements and new/changed code files, which means s/he confirms all related and removes all non-related requirements or code files. The validated requirements are automatically linked to the work item and the validated code files are stored in a new revision in the VCS. A new revision artifact is automatically created and stored in UTC and linked to the work item.

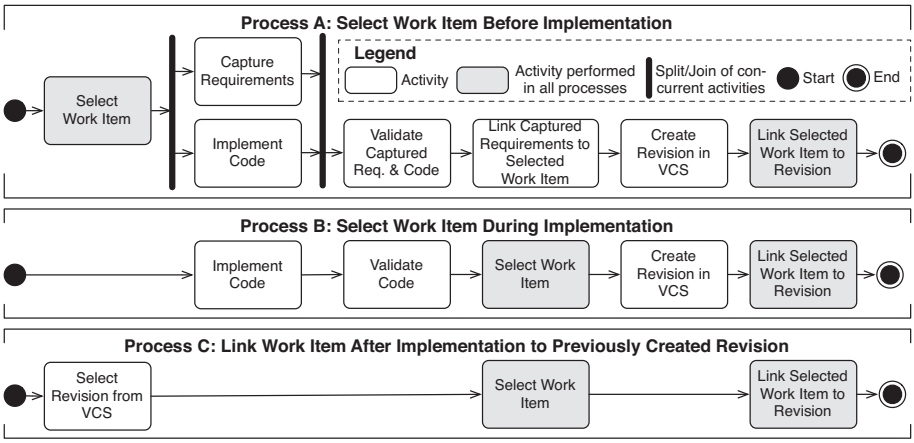


Figure 1: Traceability Link Creation Processes

2.1.2 Process B) Select Work Item During Implementation

In contrast to Process A, in Process B a developer does not need to select a work item before implementation. Instead, s/he starts directly with implementation. After the implementation of code and before creating a new revision stored in the VCS, the developer validates the new/changed code files and selects a work item from his/her list of assigned work items. A new revision with the validated code files is stored in the VCS. A new revision artifact is automatically created and stored in UTC and linked to the selected work item. In this process, no requirements are captured and validated.

It is important to note that Processes A and B do not force developers to select a work item related to the current implementation. In case the developer implemented code that s/he does not want to be linked to a work item, s/he can omit the linking of a work item, which ends Processes A and B.

2.1.3 Process C) Link Work Item After Implementation

In contrast to Processes A and B, Process C occurs after implementation and it represents an alternative way for the developer to link code to a work item. A VCS stores a history of all previously created revisions with information by whom and when each revision was created, as well as all changed code files. In case a developer has implemented code without selecting a work item before implementation (see Process A) or without selecting a work item during implementation (see Process B), s/he can manually select to link a previously created revision to a work item from his/her assigned work items list. A new revision artifact is created and stored in UTC and linked the selected work item. Like in Process B, no requirements are captured and validated.

Issue tracking systems (e.g. Trac) and project management applications (e.g. Redmine) also support VCS integration like UTC, which means linking work items to revisions. However, these tools do not support requirements as discrete artifacts. Tools supporting the same artifacts as UTC are, for example, IBM Rational Team Concert or Polarion Requirements. However, unlike these tools, UTC can capture links between these artifacts during development. Moreover, UTC can automatically infer direct traceability links between requirements and code using work items, which is explained in the following.

2.1.4 Inferring Traceability Links Between Requirements and Code

The created traceability links of Processes A, B and C are used by UTC to infer direct links between requirements and code based on the corresponding work items. In [DNP12], we presented an algorithm for inferring links that is executed when the developer changes the completion status of a work item from *assigned* to *done*. The algorithm connects in a brute force manner all linked requirements of a work item with all the code files in the linked revisions of a work item.

Changes in the code do not have a direct impact on the artifacts and links stored in UTC. However, changes in the code can lead to new (inferred) links to requirements. Over time, inferred links might become obsolete due to work on other work items. Thus, we are currently working on intelligent algorithms to discard links not relevant anymore.

Summing up, the only manual work in UTC is to establish initial links between work items and requirements (which is typical for issue management) and to validate the captured links (which should be easy as the links refer to the work just finished). Besides this, there is no other additional work required to achieve traceability between requirements and code.

2.2 Features for Traceability Link Usage

Versioning: The EMFStore [EMF] is a repository and VCS for the Eclipse Modeling Framework designed for collaborative editing and versioning of models. The EMFStore is the foundation for UNICASE, and thus UTC. All artifacts in UTC are part of a model that is versioned with EMFStore. This allows versioning all artifacts and the traceability links between them and as a result, supports merging and conflict detection. For example, one can follow all changes of a requirement and its traceability links over time as well as revert to a previous version.

Graph Visualization: UTC supports graph visualization of all artifacts and the traceability links between them. Advanced layout algorithms can be applied to the graph and one can search within the graph.

Traceability between Requirements & Code: Using inferred links between requirements and code, UTC helps to analyze which code contributes to the realization of which requirement.

Requirements Context: During implementation, a developer can look at the requirements context that shows all requirements linked to the currently open code file. Due to agile software development techniques, development teams can change quickly. This feature supports new developers joining the project trying to understand the purpose of the implemented code.

Progress of Implementation: Work items have a completion status and are linked to requirements. Thus, work items enable to identify not implemented requirements as well as the progress of their implementation. UTC enables to see how far all requirements are already implemented, as well as identifying not implemented requirements requiring increased attention.

Requirements Impact Analysis: If a requirement needs to be changed to reflect changed customer demands, all related artifacts potentially affected by this change can be identified. Affected requirements and work items can be identified, e.g. if a change in a requirement is comprehensive, related requirements and their planning of realization described in work items needs to be adapted. An initial set of code files can be identified, which can be a starting point for detailed impact analysis. The changes in the code files can result in additional changes in other code files.

3 Conclusion

We developed UTC and it integrates requirements engineering, project management and code implementation in a single environment with full traceability between all artifacts. Currently UTC is used in an academic case study. We want to compare the effort and quality of the created traceability links to the results of other conducted exploratory case studies. Furthermore, we will work on intelligent algorithms to discard links between requirements and code not relevant anymore due to work on other work items.

References

[DNP12] Delater, A.; Narayan, N.; Paech, B.: Tracing Requirements and Source Code During Software Development. In ICSEA 12: Proc. 7th Int. Conf. of Software Engineering Advances, Lissabon, 2012; pp. 274-282

[EMF] EMFStore, A model repository for EMF-based models, http://eclipse.org/emfstore/ (Last Access: January 30, 2013)

[UNI] UNICASE, Technical University Munich, Chair for Applied Software Engineering, http://www.unicase.org/ (Last Access: January 30, 2013)

[UTC] UNICASE Trace Client at Google Code, http://code.google.com/p/unicase/wiki/ TraceClient (Last Access: January 30, 2013)

ZeMoSS – Zertifizierung und modellgetriebene Entwicklung sicherer Software

Defining requirements on domain-specific languages in model-driven software engineering of safety-critical systems

Michael Wasilewski[1], Wilhelm Hasselbring[2], Dirk Nowotka[3]

[1]Vossloh Locomotives GmbH, 24152 Kiel
http://www.vossloh-locomotives.com/
[2]Kiel University, Dept. Computer Science, Software Engineering Group, 24118 Kiel
http://se.uni-kiel.de/
[3]Kiel University, Dept. Computer Science, Dependable Systems Group, 24118 Kiel
http://zs.uni-kiel.de/

Abstract: Domain-specific languages are designed and used to assist software development in various domains. Safety-critical systems such as aviation systems, railway control systems and nuclear power plants require certified software by law. This paper focuses on domain-specific languages that are used to represent a physical reality and to describe the behavior of a control software as a finite state machine. Furthermore we focus on domain-specific languages that are able to generate source code for sensor/actor systems from a specified finite state machine model. The source code is intended to be compiled and operated in a fixed time slot of a real-time operating system of a safety-critical controlling hardware.

We give an example of a model that is expressed using a functional tree, a method that is based on input and state space partitioning. We show that models expressed by a functional tree are equivalent to deterministic and complete finite state machines. To formally prove the equivalence we analyze a model in terms of automata theory. We will furthermore show that omitting the properties of determinism and completeness violates normative requirements when a model is used to generate software for safety-critical systems.

The major contribution of this paper is the definition of formal requirements on domain-specific languages employing formalisms of automata theory. The requirements are easily verifiable criteria for domain-specific languages to assess the suitability in an engineering process of a safety-critical system. We analyze two example modeling languages for their suitability to create a source code for safety-critical applications.

1 Introduction

Model-driven software development uses a formal description of a model and code generators to obtain an executable software for specific purposes. The base for a formal description of a model can be either a domain-specific language (DSL) or a universal modeling language such as the UML. The design goal for a DSL is to cover specific problems and properties of its domain such as track topologies and signaling layouts for railroad infrastructures [GHH+12]. Increased attention has to be payed if DSLs are used to create

source-code to be used in safety-critical applications. Design properties of a DSL can lead to a risks for legal assets like life, healthy and property.

Our goal is to define requirements for domain-specific languages from which source code is generated for safety-critical applications. In Section 2 we present the scope of this paper. We will illustrate the use of a Functional Tree (FT) and Finite State Machine (FSM) as two possible ways to express a model of a signal processing system. In Section 3 we will give a brief introduction into expressing a model using FTs with reference to [WH11].

We will compare the expression of a model as FT and show the weaknesses of the equivalent expression as FSM. With an example model we show a simple method to obtain an alphabet from physical states and inputs in order to transform a model expressed by FTs into an automaton. We analyze in Section 4 which properties of a automaton are needed to represent a model as a FSM with the same formal rigor as a representation with a FT provides.

A formal definition of an automaton is introduced in Section 5 and an automaton type is selected for our goal to define formal requirements for a DSL. The analysis in Section 4 is the base for defining requirements on DSLs in Section 6. We will use the formalisms of automata theory to define requirements on a domain-specific language as the contribution of this paper. We will compare the defined requirements to legal requirements such as normative regulations for the certification of software in safety-critical systems and show potential violations.

Finally in Section 7 we analyze two languages based on our defined requirements. As examples we use the DSL MENGES [GHH+12] as a representative of a highly specialized DSL. As representative for a general-purpose language we analyze state charts in the UML. We show the weaknesses and which additional effort is needed for a suitable use of these languages for modeling of safety-critical systems.

2 Scope

The scope of this paper is the development process for the software of a safety-critical system as in Figure 1.

We assume that functional requirements for the software are defined and a DSL is used for their formal expression. We furthermore assume that a selected DSL's output model can be expressed as a FSM or a FT. For both options a specific code generator can create a source code that is passed to a target specific compiler. The finally generated software can be operated in a safety-critical system. Our goal to define formal requirements on a DSL for safety-critical systems is achieved with the following steps

1. We give an example and analyze (see 1 in Figure 1) the properties of a model that is expressed using a FT
2. We transfer the analyzed properties to a model that is expressed as a FSM and show that models without the analyzed properties violate legal requirements for software

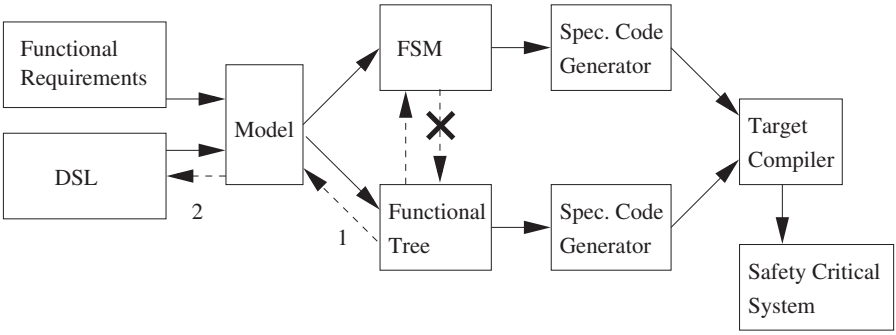


Figure 1: Scope of analysis

in safety-critical system

3. We define formal requirements on a DSL for model-driven software engineering of safety-critical systems to create a model such that legal requirements are not violated,(see 2 in Figure 1)

The reason for the choice of FTs is the method's strength that formal uniqueness and completeness of a designed model is implicitly given if it can be expressed by a FT. This property is given by the mapping of physical values as intervals on defined input and state space partitions of a model. Formal uniqueness and completeness of expressing functions or models by FTs are proven with the set theory. The formalism of this method was first time introduced in the context of a rail vehicle project in [WH11]. We will show that a model expressed by a FT has an equivalent expression by a FSM, but only a limited class of FSMs can be expressed by a FT (see Figure 1). We will furthermore show that the classes of FSM's that cannot be expressed by FTs violate normative requirements for safety-critical software.

3 Background and formal basis

In a first step we design an example model of a function for the control of a diesel engine as in Figure 2. We use a FT as expression method to explain the background of this paper. We will present an equivalent expression of our designed model as FSM to show the weaknesses of a FSM expression. Our goal is to find the differences of both expression methods by an analysis of our model as an automaton. We will introduce the formal definitions needed for the analysis of the designed model in terms of automaton type and an alphabet. Finally we will analyze the properties of our model considering the following aspects

- What kind of automaton is created if it is expressible by a FT ?

Variable	Value	Meaning
S (discr.)	S_{halt}	Engine halted
	S_{start}	Engine starting
	S_{run}	Engine running
	S_{stop}	Engine stopping
	S_{Ω}	Unknown state
AS (bool)	TRUE	Switch ON
	FALSE	Switch OFF
ST (bool)	TRUE	Start
	FALSE	No Start
n_{rpm} (cont.)	R_{LO} : $n_{rpm} < 1$	Engine halted
	R_0 : $1 \geq n_{rpm} < 400$	Engine turning
	R_{HI} : $n_{rpm} \geq 400$	Engine running

Table 1: Variable partitions

- Which is the alphabet of the automaton ?
- Which language is accepted by the automaton ?
- Which are the differences of a model expression by a FSM and by a FT ?

3.1 Example model design and functional trees expression

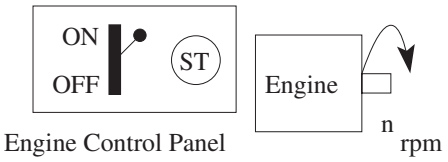


Figure 2: Example of an Engine Control

The system in our example in Figure 2 consists of an Engine Control Panel and the engine. The input values for our model are the activation switch (AS) and the starter button (ST) on the Engine Control Panel and the engine speed (n_{rpm}). Our design starts with the decisions which input and state values we use in our model and how the input and state values are represented and divided into partitions. For an expression of the model by a FT the partitioning is the basic step.

We will design our model such that the starter button (ST) and activation switch (AS) are represented by boolean values with the partitioning into TRUE/FALSE. The engine speed (n_{rpm}) is represented by a continuous value with a partitioning into intervals. We will furthermore define that our engine can be in the states S_{halt} if the engine is halted, S_{start} if the engine is commanded to start, S_{run} if the engine is running and S_{stop} if the engine

is commanded to stop. For any other state we define S_Ω as the 'unknown state'. The input and state variables and their partitions for our model are shown in Table 1. To design the model of our engine control function we have to define the conditions for the changes of the state variable S from one of it's partitions into another one. This step is equivalent to the design of a FSM and the definitions of the transitions from one state to another. To show the capability of using FTs as an expression for our model we use an example as in Figure 3.

This FT shows an example of a model which starts by processing state variable S with all of it's partitions as in Table 1. The partitions are represented by the edges of the tree. Depending on the partitions and the values of subsequent variables the edges are taken to the subsequent nodes (circle) until a leaf (square) is reached. The leafs (squares) of the tree contain the states into which our model changes after one computation step. So every path in this FT is equivalent to a transition of a FSM.

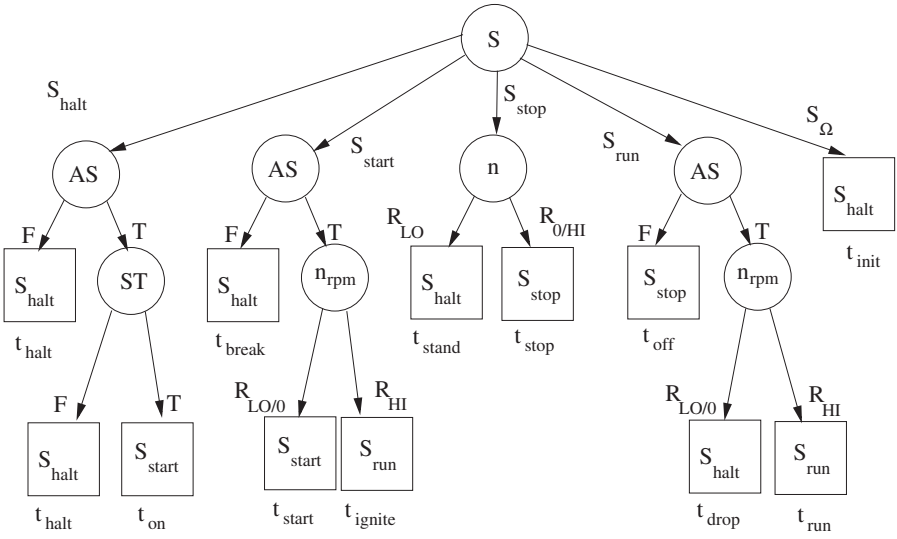


Figure 3: Functional tree for Engine Control

We can see that the different paths in our FT depend on different input variables and different processing orders. This is equivalent to different conditions for a transition of a FSM. For example the transitions t_{stand} and t_{stop} are valid for a stopping engine (S_{stop}) and only depend on the speed (n_{rpm}) to determine if the engine is already stopped ($n_{rpm} \in R_{LO}$) or still stopping($n_{rpm} \in (R_0 \cup R_{HI})$). As second example the transitions t_{halt} and t_{on} are valid for a halted engine (S_{halt}). It is started (t_{on}) or remaining in standstill (t_{halt}) depends on the inputs of the activation switch (AS) and the start button (ST). The conditions for all transitions of this model can be found in Table 3 when we analyze our model.

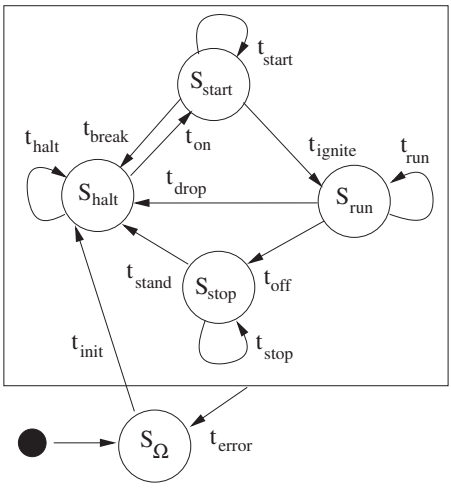


Figure 4: Finite State Machine for Engine Control

Our example shows that if a variable is considered in a FT to determine a result then it is considered with all it's partitions (completeness). If a variable is not considered in a FT then the result is the same for all of it's partitions (independence). The characteristic property of a FT is that every value of a variable has to be and can only be element of exactly one of the variable's partitions. For the expression of a model with a FT it means that every combination of input and state space corresponds to exactly one path in the FT. The consequence is that only models can have an expression as a FT which correspond to a deterministic and complete automaton. We will formally prove this property in this paper.

As shown in Figure 1 we assumed that a model expressed as FT can be also expressed as FSM. For our example, the FT in Figure 3 is equivalent to the FSM in Figure 4. Note that the expression as FSM shows the transition t_{error} to represent an unknown error that can corrupt the state variable of the model (e.g. bad memory access, hardware error etc.). This transition is not shown in the FT. The reason is that both expression FT and FSM cannot specify a defined condition for this transition. Important is that both expressions show the same reaction of the model if the state variable is in the 'unknown state' S_Ω

The expression as FSM shows the states and the transitions of the model as shown in Table 3. The conditions for every transition can be defined, but there is no automatic mechanism to ensure that all combinations of input signal are be processed by the FSM in every state. There is also no automatic mechanism that ensures that determinism for the transition is specified. To ensure uniqueness and completeness of a FSM additional verification effort is required in the design process while using a FT implies these properties. If a DSL is involved in the model design for a safety-critical system we demand that the DSL's output is a formally complete and unique model.

With FTs a designer has to start the modeling process with a correct partitioning of input and state variables for a model. Extensions, refinements and changes of the model design are expressed as changes in the paths of the FT for the model. The properties of FTs as expression method for a model ensure that for all combinations of input and state variables the designed model is formally unique and complete. The method gives a designer the choice in which context physical values are used to compute a result or not. This makes the method easily scalable for higher numbers of physical values. Verification efforts and finally development costs are only limited to verify the correctness of the designed model to meet the functional requirements, as shown in in Figure 1.

The verification of correctness of a model expressed as FSM is often limited to only cover the functional requirements. A verification of formal uniqueness and completeness of all input and state value combinations is often not done. The reason is that no selection mechanisms for relevant combinations of variables and states exists comparable to the partitioning mechanism of a FT. A non-deterministic FSM can be in theory transformed into a deterministic FSM by using it's power set. Completeness and determinism can be verified then, but this will increase verification effort and development costs significantly.

Our model is intended to be used safety-critical applications. Mistakes such as incomplete or contra-dictionary definitions during the model design can result in severe hazards for human life or material assets. Normative regulations such as [CEN01b](see 8.4.2), [IEC11](see 7.2.2 and 7.4.5) require a unique and complete functional specification of software for safety-critical applications. We have demonstrated that expressing a model with a FT is limited to models that correspond to a deterministic and complete automaton, but this is exactly the type of model we need to meet the requirements of software in safety-critical systems. Our goal will be therefore to define formal requirements for a DSL such that the output model will have a defined behavior for every combination of inputs and states and be expressible as a FT.

3.2 Functional tree expressed model as automaton

For the statement of formal requirements on DSLs we will analyze a designed model that we expressed using a FT as an automaton. The automata theory uses sets of states and symbols to describe alphabets and languages that are accepted by an automaton. The basic elements of FTs are disjoint sets to describe the processed signals. We will connect both concepts.

In a first step we will describe the relation between the disjoint sets of value partitions as used for a FT and the sets of input symbols of an automaton, as in Table 2. We have listed all input space partition combinations of our example model in Figure 4. Every input space partition combination is assigned to one symbol s_1 to s_{12}. We obtain a bijective function between our model's input space partition combinations and the symbols of an formal language. This approach can be extended to any number of input variables such that we obtain a partitioning of the input space of a model and one corresponding language symbol for every partition. Similarly the partitions of one or more state variables provide

a partitioning of the states space of a model. Every partition of the state space of the model will correspond to one state of an automaton.

The analysis of a model that is expressed as a FT is based on the input space $\mathbb{I}$, the state space $\mathbb{S}$ of the model and a definition of functional independence. The input and the state spaces are defined based on the signals that are processed in the computation (note that variables over continuous domains are discretized by considering intervals):

$$\mathbb{I} \subseteq (\mathbb{R} \times \mathbb{R})^i \times \mathbb{Z}^j \times \mathbb{B}^k$$

$$\mathbb{S} \subseteq (\mathbb{R} \times \mathbb{R})^{i'} \times \mathbb{Z}^{j'} \times \mathbb{B}^{k'}$$

where $\mathbb{I}$ and $\mathbb{S}$ are finite and i, i', j, j', k, k' are naturals. By common notation, $\mathbb{R}$ denotes the set of reals, $\mathbb{Z}$ denotes the set of integers, and $\mathbb{B}$ denotes a binary set. Our example in Table 1 uses just one state variable, namely S. We use three signals in our example, namely AS, ST and n_{rpm}.

AS	ST	n_{rpm}	Symbol
OFF	T	R_{HI}	s_1
OFF	T	R_0	s_2
OFF	T	R_{LO}	s_3
OFF	F	R_{HI}	s_4
OFF	F	R_0	s_5
OFF	F	R_{LO}	s_6
ON	T	R_{HI}	s_7
ON	T	R_0	s_8
ON	T	R_{LO}	s_9
ON	F	R_{HI}	s_{10}
ON	F	R_0	s_{11}
ON	F	R_{LO}	s_{12}

Table 2: Symbol table

For the specification of a function in our context we use a set of n variables x_0 to x_n of continuous, discrete or binary type. A function of an automaton (e.g. transition, output) will be defined as $f(x_0 \dots x_n)$. We define functional independence of a function $f(x_0 \dots x_k \dots x_n)$ from the variable x_k if the following holds:

$$\forall x \colon f(x_0 \dots x_k \dots x_n) = f(x_0 \dots x \dots x_n)$$

This definition of functional independence is required for a formally correct split of the input and state space $\mathbb{S} \times \mathbb{I}$ into partitions. The connection of a FT as in Figure 3 and the symbols, states and transitions of an automaton is shown in Table 3. Each row of the table represents one path of the FT. We can also see that each row represents one of the partition P_1 to P_{12} of our model's input variable (AS, St, n_{rpm}) and state variable (S) space.

Considering the functional table 3 as state and transition table of an automaton we can find the automaton states as partitions of the state variable S of the functions tree. For comparison see Figure 4. The automaton transitions correspond to paths of the FT. Note that the transition t_{halt} is mentioned two times in Table 3 and two times in the FT, see Figure 3 while it is only mentioned once on the FSM expression of the model, see Figure 4. Referring to our example model the technical meaning is that the start of the engine can be blocked by two conditions, namely the activation switch turned off (AS=OFF) or the starter button not pushed (ST=F).

Not all transitions from one state to another depend on all input variables while all input variables have to be considered for a complete and unique design of the model. The $\forall$ symbol in a row means that an input variable has not to be considered, as the final result is the same for any of it's values. In the FT expression of a model the corresponding variable

Partition	State S	Inp. AS	Inp. ST	Inp. n_{rpm}	Transition	Symbol
P_1	S_{halt}	OFF	$\forall$	$\forall$	t_{halt}	s_1,s_2,s_3 s_4,s_5,s_6
P_2	S_{halt}	ON	F	$\forall$	t_{halt}	s_7,s_8,s_9
P_3	S_{halt}	ON	T	$\forall$	t_{on}	s_{10},s_{11},s_{12}
P_4	S_{start}	OFF	$\forall$	$\forall$	t_{break}	s_1,s_2,s_3 s_4,s_5,s_6
P_5	S_{start}	ON	$\forall$	R_{HI}	t_{ignite}	s_7,s_{10}
P_6	S_{start}	ON	$\forall$	$R_{LO/0}$	t_{start}	s_8,s_9,s_{11},s_{12}
P_7	S_{stop}	$\forall$	$\forall$	$R_{0/HI}$	t_{stop}	s_1,s_2,s_4,s_5 s_7,s_8,s_{10},s_{11}
P_8	S_{stop}	$\forall$	$\forall$	R_{LO}	t_{stand}	s_3,s_6,s_9,s_{12}
P_9	S_{run}	OFF	$\forall$	$\forall$	t_{off}	s_1,s_2,s_3 s_4,s_5,s_6
P_{10}	S_{run}	ON	$\forall$	R_{HI}	t_{run}	s_7,s_{10}
P_{11}	S_{run}	ON	$\forall$	$R_{LO/0}$	t_{drop}	s_8,s_9,s_{11},s_{12}
P_{12}	S_{Ω}	$\forall$	$\forall$	$\forall$	t_{init}	s_7,s_8,s_9 s_{10},s_{11},s_{12}

Table 3: Functional table

is not in the path that represents the transition. For example the transition t_{halt} of partition P_2 is independent of the variable n_{rpm}. In the FT of the model Figure 3 the variable n_{rpm} is not in the corresponding path. Considering the model as an automaton we need to select the input symbols for transition t_{halt} such that we cover the corresponding input space partitions as in Table 2. In our example the symbols s_7,s_8,s_9 have all in common the input variables AS=ON and ST=F while all together cover all partition of n_{rpm}.

With the definition of functional independence we can use the symbols as in Table 2 for a formally correct transformation of a model expression as FT into a model expression as an automaton. Our assumption is proved that every model expressed as FT can be expressed as a FSM. For the partitions P_1 to P_n of the FT and the input and state space of our model $\mathbb{S} \times \mathbb{I}$ holds:

$$\bigcup_{k=1}^{n} P_k = \mathbb{S} \times \mathbb{I}$$

Furthermore for every partition of the state variable S we can find all input symbols of the automaton as in the Symbol column in Table 3 exactly once. The formal definition of functional independence says that not considering a variable in a computed result is equivalent to a result that hold for all values of the variable. This proves that an automaton that corresponds to model expressed as FT is deterministic and complete.

4 Properties of an automaton obtained via functional trees

Our analysis starts on the base of a transition system (TS) as defined in [BK08], Chapter 2. We focus on the states and transitions of an automaton so the mentioned TS is an appropriate base. We will consider the processing of a new input symbol as a transition from one state to another state.

The analysis is based on the assumption that an automaton is transformed into source code that is finally compiled and executed in a safety-critical system. The execution of the source code is steadily repeated in fixed time slots and potentially never terminates. We will analyze the properties of an automaton that is obtained by designing a model using a FT. The goal is to express the found properties in terms of automata theory [Sak09].

A FT model creates an automaton processing a finite length input
: This property results from the fact that the automaton created by a FT runs for a potentially infinite time. However in every processing cycle a new symbol of the input space is processed which corresponds one transition of the automaton. The finally processed sequence of input symbols keeps a finite length in every processing cycle.

The automaton is deterministic
: The determinism of the automaton is given by the uniqueness property of the FT. For *each* input symbol and state there is at most one transition. The automaton has therefore at most one transition for each input element at each state.

The automaton is complete
: The completeness of the automaton is given by the completeness property of the FT. For *every* input symbol and state combination there is exactly one transition. A preserved state is expressed by a reflexive transition. The automaton has therefore exactly one transition for each input symbol. In safety-critical applications the complete behavior of an automaton has to be specified, verified and tested. That includes implicit reflexive transitions.

The automaton provides a specified result for every processed input sequence
: The FT formal base ensures that an automaton provides a specified transition for every partition of the input and state space. The output of the automaton is a specified, verified and tested function for every computation cycle. This property is required for operation in a safety-critical system.

Note that, a notion of automaton is not yet given here, but will follow in the next section. The property to provide a specified result for every partition of the input and state space of the automaton is the key point for the definition. Our automaton can, in principle, be operated with any sequence of input symbols. However it is required that the automaton's behavior is always within the specified scope. For the further analysis we therefore assume that our automaton processes any sequences of input symbols. This assumption is the base for our selection of an automaton model.

5 Selection of automaton type

In Section 4 we have obtained the properties for an automaton is the output of a model design using a FT. The analysis showed that we require a complete, deterministic automaton with output to describe the automaton investigated. A complete, deterministic automaton A with output is defined as follows (see also [AS03]):

$$\mathcal{A} = (\Sigma, X, Z, z_0, \delta, f) \tag{1}$$

where Σ denotes the set of inputs, X denotes the set of possible outputs, Z denotes the set of states, $z_0 \in Z$ denotes the start state, $\delta\colon \Sigma \times Z \to Z$ denotes the transition function and $f\colon Z \to X$ denotes the output function of the automaton. As usual, we extend δ on the set of all finite sequences over Σ as follows. Let $\delta'\colon \Sigma^* \times Z \to Z$ be such that $\delta'(\sigma' a, z) = \delta(a, \delta'(\sigma', z))$, where $\sigma' \in \Sigma^*$ and $a \in \Sigma$ and $z \in Z$. The output of the automaton is taken by f from the state the automaton ends in after reading a sequence σ, formally, $f(\delta'(\sigma, z_0))$. This type of automaton meets our key requirement to provide a specified result for every combination of input symbol and automaton state. This rigorous requirement result from normative regulations for software in safety-critical applications. The behavior of software has to be completely specified and evaluated for potential safety hazards. The use of the usual semantics of Mealy automata to ignore unspecified input would provide the same result, but violate the requirement for a complete specification. It is therefore not suitable for evaluation purposes of safety-critical software.

6 Requirements on DSLs for safety-critical systems

For the definition of the formal requirements on a source code generating DSL we refer to an automation as in Equation 1 of Section 5. Based on our analysis of automaton properties we define the following formal requirements for the use of DSLs and their generated models in a safety-critical system. We refer to the output of a DSL as a model as we have seen that a model can be expressed in different ways, namely FSM or FT.

A bijective function has to exist between the physical value and state space partitions and a model's input and state space partitions: A DSL has to provide a formal mechanism for the transformation of physical input and state partitions (such as intervals of sensor values, switch states, etc.) and a model's input and state space partitions. It has to be ensured that every partition of the input and state space of a model has a physical correspondence – even if the correspondence is identified as invalid.

The generated model has to be a non-terminating transducer: The generated model has to process inputs of potentially infinite length. However every received new input symbol since the start of the processing keeps the already processed word's length finite.

Partition	Physical Reality
$R_{LO} : T < -273°C$	Valid Temperature
$R_{HI} : T \geq -273°C$	Sensor fail

Table 4: Example of bijective function between partitions and physical reality

The generated model has to be a deterministic automaton: For application in a safety-critical systems the behavior of software has to have a unique description for the assessment of potential risks. For any symbol of Σ there has to be exactly one transition.

The generated model has to be complete automaton: For application in a safety-critical system the behavior of software has to have a complete description for the assessment of potential risks. For every symbol of Σ there has to be an element in $\delta : \Sigma \times Z \rightarrow Z$. The state machine is in continuous computation in a running system. We also require the the model for all states and inputs to be provide a specified result to be suitable for the use in safety-critical systems.

These requirements are based on the established formalism of automata as theoretical foundation and can be easily verified in practical use. Our defined requirements can be well justified by normative regulations that are demanded by law for software in safety-critical systems [CEN01b, IEC11]. The bijective function between physical input and state space partitions and automaton symbols and states is a formal criterion for the transformation of a physical reality into a model.

An example for such a transformation is the representation of a temperature in °C as signed integer as in Table 4. A value $T \geq -273°C$ represents a valid temperature measured by a sensor as physical reality while a value $T < -273°C$ is the physical reality of a sensor fail. We want to evaluate for every partition of our model whether the input and state values have a physical sense. We want furthermore specify a reaction of the software for every partition we identify as physically invalid, inconsistent or erroneous.

If a DSL's output is an automaton that ignores an input (such as a Mealy automaton can do) it violates the legal requirement for a complete specification. If the DSL's output is a non-deterministic automaton it violates the legal requirement for a unique specification that can be evaluated for safety hazards. The safety of software should not rely on potentially incorrect expectations about a physical reality and erroneous model design. Our approach automatically covers effects of component failures (such as invalid sensor data, unexpected data memory changes, etc.) that have to be evaluated during the system development process [CEN01a]. The computation of a model itself has to be specified, verified and tested with the required rigor for safety-critical systems.

The requirement for a non-terminating transducer results from the potentially infinite runtime of an embedded system. However, in practice the time between a system start and any processing cycle will be finite. This justifies to considers a signal processing system

Listing 1: MENGES example

```
process SwitchChange{ start check_lock;
   condition check_lock{
      case this.locked: release_lock()
      default: start_SwitchMotion() }
   action start_SwitchMotion() {
      ...
      continue SwitchCheckPosition}}
```

as an automaton processing words of a finite length avoiding problems with processing infinite formal objects.

7 Property analysis of existing modeling languages

We analyze two representatives of modeling languages based on our formal requirements for DSLs being used in safety-critical applications. One modeling language (MENGES project) is designed highly specific for the domain of rail signaling. Another example is UML which serves as a general-purpose language to describe models.

7.1 DSL of the MENGES project

The DSL of the MENGES project [GHH+12] was designed to describe rail signaling infrastructures. This DSL is able to describe the topology of a specific installation based on object instances of "interlocking elements" such as switches. The communication between interlocking elements is based on a role model and defined interaction protocols. The logic of interlocking elements is specified using finite state machines or processes. The elements to describe a logic are actions and conditions.

For our analysis we take a closer look at a process at the example in Listing 1. This example describes the process to change the position of a switch. It starts with a check if the switch is locked. A locked switch triggers an action to release the switch otherwise the motion of the switch is started. The motion is not mentioned in detail here. The process terminates and a following process to check the switch position is started.

The presentation paper of the DSL of the MENGES project [GHH+12] mentioned that a process can block during evaluating a condition. To realize a non-blocking process the definition a **default**-action is required. We have considered a process as a FSM. The evaluation of conditions is equivalent to the processing of different input symbols. Our analysis showed that the possibility of a process to block is equivalent to the processing of an unspecified input. The requirement for a total finite state machine was violated. Our analysis resulted in a change of the DSL design such that a **default**-action is mandatory.

Based on our formal requirements for DSLs in safety-critical systems, we have identified the model design as the critical point if the DSL of the MENGES project is used in a software development process as in Figure 1. The challenge is a suitable choice of states and input to represent rail infrastructure elements. The requirement for a bijective function between the real rail infrastructure elements and their representation in a software is a possible point for future work. Requirements for determinism, a total transitions function and specified processing of all input and state space partitions can be fulfilled, but the verification of these properties is not withing the scope of the language. Also this is a challenge for future work.

7.2 State charts in the UML

The Unified Modeling Language [Oes12] is able to specify hierarchical state machines as State charts. State charts contain states and transitions and represent an object life cycle. Actions can be defined for state entry, exit and persistence in a state. Transitions are only labels and can be described as external, implicit transitions or as so called “guards” to detect a defined conditions.

UML was designed to be processed by other tools such as code-generators to transform the models into other structures. This language was not explicitly designed to assist software development for safety-critical applications, but is however used for this purpose. Additional effort such as definitions of subsets of the language and verification techniques are needed to achieve a suitable usability of UML to assist the development of safety-critical software. Our formal requirements can be criteria for a rigorous check of language subsets or designed models. Future work can show how the formal requirements are realized with UML.

8 Related Work

We have defined our requirements on a domain-specific language for safety-critical systems without using a formal language. The problems of obtaining complete models for control systems and the limits of language parsing finite state automata are discussed in [WWW04]. This work identifies the problems without formalization, but in the context of the tool StateWorks. Our work uses a formal definition of completeness based on the set theory [WH11].

Different to [Lat03] and [KL06] we do not express “safety properties” as properties of a formal language. We use the automata theory as in [Sak09] with focus on computation mechanisms and the ability of a FSM to provide a specified result for any input. For the suitability in safety-critical systems we require a FSM to generate an output that is considered safe for every computation step. Our approach to achieve a safe behavior in a project context is based on a formally complete specification of a model providing a specified output.

Similar to purposes of runtime verification [FFM09, BGHS04] and model-checking [BBB+04, BK08] our work is focused on software with a potentially infinite runtime. We however use different different types of automata for the design of a FSM model than for the verification. Runtime verification and model-checking require decisions whether formally expressed properties are fulfilled or not. Such decision making processes and algorithms are based on infinite formal expressions and Büchi automata [Büc62]. The focus of our work is the design of FSMs. Our intention is to introduce well-researched formalisms, properties and methods for domain-specific languages to support the design of FSMs for safety-critical systems in industrial settings.

9 Conclusions and Outlook

The base of our investigations was the design of model with the formalized method of functional trees, that is already applied in industry [WH11]. We shown that a designed model with functional trees has an equivalent expression as FSM, but is more suitable for the use in safety-critical applications. The rigorous formalism of functional trees ensures that for every combinations of input and state values the model provides a specified, verified and tested result.

We have demonstrated a method to express a functional tree designed model as an automaton with an alphabet and states. The properties of the automaton are the base to define formal requirements on domain-specific languages for model-driven software engineering safety-critical systems. Our analysis showed that a model has to be able to provide a specified result for every element of it's input and state space. If a domain-specific language's model output is a FSM it has meet the following formal requirements:

- There has to be a bijective function from physical input intervals and states to model inputs and states – a simple realization is possible by space partitioning.
- The FSM has to be non-terminating, deterministic and complete. A rigorously specified, verifiable and testable reaction on invalid, erroneous or inconsistent input is required.

Our defined requirements are based on well-understood properties and an established formalism: automatons and their expression as FSMs. The key point is that these requirements are easily verifiable. With our defined requirements, existing DSLs and modeling languages can be evaluated whether they are suitable for the use in the development process of safety-critical systems. For the design of future DSLs these requirements can be a base and existing DSLs can be evaluated for their suitability using our criteria. Future work furthermore addresses analyzing the impact of our work on model checking [BBB+04] and source code instrumentation [FHRS07, vHKGH11] for runtime verification techniques.

References

[AS03] Jean-Paul Allouche and Jeffrey O. Shallit. *Automatic Sequences - Theory, Applications, Generalizations*. Cambridge University Press, 2003.

[BBB+04] R. Buschermöhle, M. Brörkens, I. Brückner, W. Damm, W. Hasselbring, B. Josko, C. Schulte, and T. Wolf. Model Checking – Grundlagen und Praxiserfahrungen. *Informatik-Spektrum*, 27(2):146–158, April 2004.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-Based Runtime Verification. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[Büc62] J. R. Büchi. Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science 1-11. In Nagel et al., editor, *On a decision method in restricted second-order arithmetic*. Stanford Univ. Press, 1962.

[CEN01a] CENELEC. *EN50126 - Railway applications: the specification and demonstation of Reliability, Availabity, Maintainability and Safety*. CENELEC, 2001.

[CEN01b] CENELEC. *EN50128 - Railway Applications: Software for Railway Control and Protection Systems*. CENELEC, 2001.

[FFM09] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime Verification of Safety Progress Properties. In *Runtime Verification 2009*, Lecture Notes in Computer Science, Grenoble, France, June 2009.

[FHRS07] Thilo Focke, Wilhelm Hasselbring, Matthias Rohr, and Johannes-Gerhard Schute. Instrumentierung zum Monitoring mittels Aspekt-orientierter Programmierung. In *Tagungsband Software Engineering 2007*, volume 105 of *LNI*, pages 55–58, 2007.

[GHH+12] Wolfgang Goerigk, Wilhelm Hasselbring, Gregor Hennings, Reiner Jung, Holger Neustock, Heiko Schaefer, Christian Schneider, Elferik Schultz, Thomas Stahl, Reinhard von Hanxleden, Steffen Weik, and Stefan Zeug. Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke. In Stefan Jähnichen, Axel Küpper, and Sahin Albayrak, editors, *Software Engineering*, volume 198 of *LNI*, pages 119–130. GI, 2012.

[IEC11] IEC. *IEC 61508 - Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES)*. IEC, 2011.

[KL06] Orna Kupferman and Robby Lampert. On the construction of fine automata for safety properties. In *In Proc. 4th ATVA, LNCS 4218*, pages 110–124, 2006.

[Lat03] Timo Latvala. Efficient Model Checking of Safety Properties. In *In Model Checking Software. 10th International SPIN Workshop*, pages 74–88. Springer, 2003.

[Oes12] Bernd Oestereich. *Analyse und Design mit UML 2.5*. Oldenbourg Verlag, 10th edition, 2012.

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[vHKGH11] André van Hoorn, Holger Knoche, Wolfgang Goerigk, and Wilhelm Hasselbring. Model-Driven Instrumentation for Dynamic Analysis of Legacy Software Systems. In *Proceedings of the 13th Workshop Software-Reengineering (WSR 2011)*, pages 26–27, May 2011. (Softwaretechnik-Trends 31(2) (May 2011) 18–19).

[WH11] Michael Wasilewski and Wilhelm Hasselbring. A Formal and Pragmatic Approach to Engineering Safety-critical Rail Vehicle Control Software. In *Software Engineering*, volume 183 of *LNI*, pages 99–110. GI, 2011.

[WWW04] Ferdinand Wagner, T. Wagner, and Peter Wolstenholme. Closing the Gap Between Software Modelling and Code. In *ECBS*, pages 52–60. IEEE Computer Society, 2004.

Opening up the Verification and Validation of Safety-Critical Software*

Hardi Hungar and Marc Behrens
Institute of Transportation Systems
DLR, Braunschweig, Germany
{hardi.hungar,marc.behrens}@dlr.de

Abstract:
Smooth cross-border rail traffic is of important interest to commercial realizations of ETCS[1]. Starting from the hypothesis that the traditional way of developing software for safety-critical systems might be an obstacle to standardizing rail traffic, the ITEA 2 project openETCS has set out to pursue the idea of transferring an open-source development style to this domain, taking the EVC[2] as a target. The goal is to formalize the requirements in a functional model, derive, via design models, an implementation, and demonstrate how the verification and validation activities necessary for certifying a resulting product could be performed. All of this is to be done as an open-source project, employing only open-source tools. One of the main motives behind the approach is to use the potential of an open community to detect design and implementation flaws much earlier than the resource-limited inspection in a traditional development setting.

This papers discusses the challenges this new approach faces from the legal requirement of adhering to the standards, mainly the EN 50128 in this case, particularly with respect to verification and validation. This comprises the interpretation and application of the standard throughout all lifecycle phases for a open-source model-based development and qualification issues for personnel and tools.

1 Motivation

Despite ongoing efforts and investments, cross-border interoperability has not yet been achieved by the ERTMS (European Rail Traffic Management System) initiative. Two main issues are costs and diverting functional implementations. As part of the ERTMS, ETCS (European Train Control System) is the on-board system for signaling, control and train protection. Within the train, its central component is the EVC (European Vital Computer).

ETCS OBUs[3] are currently expensive if compared to existing older systems. To give specific figures, the cost for equipping 50 ICE-T in 2010 was 365 t€ per unit, with additional 100 t€ per unit for estimated software maintenance over the next 15 years [Has12].

*This research has in part been funded by the German Federal Ministry of Education and Research as part of the ITEA 2 initiative. The responsibility for the content lies with the authors.

[1] European Train Control System

[2] The European Vital Computer is the central constituent of the assembly On-Board

[3] On Board Unit, composed of the EVC and its peripherals

For full interoperability, the ETCS on-board equipment has to be backwards compatible with all ETCS track-side equipments, regardless of the manufacturer, year of build, release version, etc. This is not yet the case for all EVCs. One of the reasons is that most requirements are not formally specified and thus open to being interpreted differently by certifying authorities. To prove instances of interoperability, extensive tests are required. If they fail, it is difficult to identify the responsible component (if the cause is not that two manufacturers have interpreted (completed) the requirement specification in consistent yet different ways)[4].

The two issues cost and interoperability are accompanied by further phenomena commonly observed in complex electronic equipment in rail applications. Software generally is error-prone, thus complex software will seldom be completely free of bugs (unless development methods are drastically changed). Thus, it will need continuous service. Typical software is experienced to contain 1 to 10 bugs per KLOC (thousand lines of code)[5] High quality, mature software may have 0.5 bugs per KLOC. Achieving a rate of less than one bug per 10 KLOC will certainly be very expensive. With the ETCS kernel having an estimated 100 to 500 KLOCs one may expect a number in the order of 50 to 1000 bugs. This does not mean that the EVC will be unsafe or "wrong". In fact, most of the bugs will never affect the system at all. But it means that there is a high probability that some issues will arise sooner or later.

Given that, and taking into account the long operation periods of rail equipment, it is to be expected that the control software will need bug fixing and certainly updates for other reasons for a long period. This results in several risks, from obsolescence of development tools to vendors going out of business. In the world of proprietary, closed systems this entails high risks (i. e. high expected lifetime costs) for the railway undertaking.

The ITEA 2 project openETCS [ITE12] approaches these problems by:

Standardization reducing the diversity of equipments and promoting one common kernel for the EVC,

Formalization of requirements removing ambiguities,

Open development starting from requirements down to the software, and including all activities within the open ecosystem.

Our paper is concerned mainly with this third step: What does V&V within an open development mean, and how could it be implemented in the regulated domain of safety-critical railway equipment?

An essential ingredient of the approach is to move into making the system *open proof* [opee]: A software or a system is "open proof" if all of the following are FLOSS (Free and/or Libre and/or Open-Source Software):

1. the entire implementation (requirements, design, code, required documentation for use/maintenance, etc.)

[4] One indication of the diversity is the high amount of change requests set to be designer's choice

[5] We give commonplace figures for motivation purposes only. The topic of software and bug metrics is too complex to be discussed here.

2. automatically-verifiable proofs of at least one key criterion, and

3. all required tools needed for use and modification of the above (e. g., compilers, editors, proof-checkers, proof assistants, etc.)

Given an open proof implementation of the software kernel, it can freely be used, studied, modified and redistributed. So bugs can be identified and corrected in an early development state. This sounds very attractive, though it is by no means already an answer to the problems faced by railways and manufacturers since e. g. the specific verification obligations and the clear separation of roles required by the EN 50128 are not reflected.

[Whe06] makes a case for the superiority of FLOSS development, which he summarizes in the sentences:

> "Normal" mathematicians publish their proofs, and then depend on worldwide peer review to find the errors and weaknesses in their proofs. And for good reason; it turns out that many formally published math articles (which went through expert peer review before publication) have had flaws discovered later, and had to be corrected later or withdrawn. Only through lengthy, public worldwide review have these problems surfaced. If those who dedicate their lives to mathematics often make mistakes, it's only reasonable to suspect that software developers who hide their code and proofs from others are far more likely to get it wrong.

Again, the argument sounds good while its range is somewhat limited. For a successful application in the rail domain, the "open" approach must be made consistent with current regulations, in our case the CENELEC standards, most prominently the EN 50128 describing how software for safety-critical rail systems is to be developed.

The standards are geared towards producing a system of proven high quality, not one that is to be expected to be changed often. This contrasts with the observations made on present-day software-intensive systems, where patches and new versions are not that rare. This manifests itself in exemplifying the development in the form of a waterfall process or in form of the V model. The process has of course provisions for maintenance updates, but does not emphasize this aspect.

And it is geared towards a company or fixed organization performing the development, with requirements on the organization and personnel structure having a clear separation in roles.

2 open Approach Versus CENELEC

In this section, we discuss problems and options for reconciling an open development approach with the existing legal framework. We present the current view we have on that, it is not an agreed position of the project consortium.

2.1 Overview of the CENELEC Requirements

The main goal of the CENELEC standards EN 50126, EN 50128 and EN 50129 is to control the safety risks caused by information-processing railway applications. Risk is defined as the product of the damage resulting from some event and its occurrence probability. As it is extremely difficult to compute the risk *caused* by some equipment, the regulations address the development process to reduce the probability of unsafe or critically erroneous applications being produced. Concerning the questions related to the suitability of an open development, mainly three different categories of regulations are relevant. We concentrate here on the software regulations (EN 50128).

Management and organization The equipment is built by an entity, called the *supplier*. Personnel responsibility for different roles in the development must be assigned, and the persons must be demonstrably qualified for their respective tasks. There are restrictions on the organizational structure, e. g. which tasks must be performed by different persons or organizations. All of the development must be planned and documented in detail.

System design The design shall be documented in a top-down form, with clearly defined steps and interfaces between those steps. As a general principle, it must be verified that each step achieves its goals, and the end result must be validated against the original requirements. Verification is a general term not to be confused with proof in the mathematical meaning of the word, neither the one of formal logic with (in principle) machine-checkable results. A particular instance of the V model illustrates an acceptable lifecycle, any deviating approach shall be established as being equally suitable. Activities to be performed in the steps and documentation to be produced are defined in detail. The adopted lifecycle must make provisions for iterations, with the general idea of achieving a consistent and consistently documented development through impact analyses and change management.

Design means Tools that are used in the design are classified according to their role according to the impact they may have on the safety of the resulting software. T1 tools like requirements editors have no direct contact to the code produced. T2 tools may be involved in checking or testing software but not produce or modify code — their failure may lead to errors not being revealed but not to create errors. T3 tools finally produce or modify code or data to be used by the code. All tools used in the design must be qualified with respect to their role, T3 tools in particular have to be validated or their results extensively checked. Furthermore, the tool set used in development must be coherent. Further restrictions concern the verification activities to be performed, e. g. structure-dependent tests. Again, a consistent set of measures must be selected, where the standard defines some accepted combinations and lists a lot of (other) techniques to chose from.

2.2 Matching openETCS

One of the goals of moving to "open" is to have the development artifacts and results inspected by a large set of people, to discover errors or things which could be solved in a better way. And then, of course, have corrections or improvements performed by the "community". We discuss where this approach collides with the process prescribed (or at least recommended) by the standard.

General The EN 50128 makes provisions for changes to a software after deployment, and does so in a sensible way (impact analysis, review of only the impacted things). But it remains in the responsibility of the supplier to organize the development so that a change can be made without too much effort. This general observation becomes more virulent when revisions are envisioned to become more frequent and come from different parties as in the open approach.

Management and organization Partly implicitly, but also explicitly, the standards assume a fixed organization carrying out the development. All plans for all activities have to be fixed, and changes to a plan entail a number of regulated steps. Personnel may of course change in the course of a project, but responsibilities have always to be clearly assigned. An open community from the very idea does not satisfy these requirements, so an organizational answer to this demand has to be found.

System design openETCS goes for a model-based development of the software. Though the 2011 version of the EN 50128 includes model-based design as a highly recommended approach for SIL-4 software and lists appropriate techniques, the decision how this can be instantiated is largely left to the entity performing the development. In particular the role models may take as or in artifacts is not defined. An explorative design style, which is very common in model-based design, with parts of the system being modeled in detail while other remain only rudimentarily considered is ad-hoc not conformant. The artifacts corresponding to the phases of the proposed (illustrative) process of the standard would have to be constructed later, when all system parts have been detailed sufficiently.

Design means Model-based development comes usually with a higher degree of formalization than traditional styles. This is in accordance with the standard, which would also accept rigid (mathematical or formal logic) proofs for verification steps. The state of the art is albeit still quite far from what Wheeler advocates as "open proof". In particular, the tools themselves while complying to open proof are far from being readily usable. Among other issues, frequent updates—what is more the rule than the exception in an open community—entail many difficulties from qualification to repeatability.

2.3 Detailing the "Open" Concept

The "Open" concept implies a high degree of freedom in terms of possibilities to contribute and review given by the hosting platform Github [opeb]. Within this freedom openETCS sets out to have publicly available proofs for the source code licensed under FLOSS[6].

General Within openETCS, the source code of the EVC is categorized as "Source Code", "Qualified Source Code", "openETCS Source Code" and "Approved Source Code". Starting out with the initial openETCS partners [opec] providing qualified trusted developers, an initial contribution of "Qualified Source Code" is deployed into the openETCS repository, making it "openETCS Source Code". This deployed contribution becomes public after an incubation phase and is then available to the public and can be picked up by a development community, which itself can contribute "Source Code" to the group of qualified trusted developers. The "Source Code", in case of approval by the qualified trusted developers, becomes the "Qualified Source Code" as part of the openETCS trusted repository and after further approval by verification and validation at the end of the development cycle becomes the "Approved Source Code" and thus ready for integration by the suppliers.

Organization The "openETCS Eco System", one of the projects running inside the openETCS container [oped], describes the development and release process and gives detailed instructions on how to participate. Two roles, the *contributor* and the *committer* are defined. While the contributor can change his forked repository and apply for his changes to be inserted into the openETCS repository by a pull request, the committer directly writing to the repository is responsible to take the decision on accepting or rejecting the pull request. As a gatekeeper to the intellectual property container on the repository, the committer needs to be aware of intellectual property rights clearance.

Process Anticipating ways to profit from the potentials of an open community, the following central characteristics for the process were identified:

Modularity of Approval by enabling small and independent proofs for verification and validation, to lower the complexity and ease review.

Executability by having automatically executable tests, to speed up testing procedures and achieve high test throughput.

Formalized Impact Analysis with reporting for each occurring issue, vital for channeling and tracing the changes to the software during the full process.

Traceability up to the requirements, mandatory for SIL-4 software development and specifically important in an open project, as one needs to trace V&V verdicts to trigger decisions for changes that have a direct impact on committed software.

[6] free-libre open source software

Means Concerning V&V, the first step is to work out the verification and validation plan covering the following central topics:

Executive Summary giving an overview of the major elements from all sections.

Problem Statement describing the challenges to be answered by V&V as well as the decisions to be taken based on the V&V results as well as how to cope with potentially faulty output. It further describes the accreditation scope based on the risk assessment done on V&V-level.

V&V Requirements Traceability Matrix links every V&V artifact back to the requirements to measure e. g. test coverage and to directly link V&V results to the requirements.

Acceptability Criteria, as the direct translation of the requirements into metrics to measure success, are used e. g. for burndown charts within the process.

Assumptions that are identified during the design of the verification and validation strategy and how these assumptions have an impact on the verdict by listing capabilities and limitations.

Risks and Impacts that come across the execution of V&V tasks together with the impacts foreseen.

V&V Design states how the V&V process builds up including data preparation, execution and evaluation.

V&V Methodologies giving a step-by-step walkthrough of all possible V&V activities including the assumptions, and verdict-relevant limitations and criteria for, e. g., model verification, model-to-code verification, unit testing, integration testing and final validation (according to the standard, this involves running the software on the target hardware).

V&V Issues describing unsolved V&V issues and their impact on the affected proof or verdict.

Peer Reviews going into details on how the community can take part and how official bodies and partners are integrated into the development and review process.

Test Plan Definition going into the details of testing by describing among other things:

Title as a unique identifier to the test plan.

Description of the test and the test-item giving information about version and revision.

Features to be tested and not to be tested in combination are listed together with information background.

Entry Criteria which have to be met by the EVC before a test can be started, e. g. that the EVC has to be in level 3 limited supervision with the order to switch to level 2.

Suspension criteria and resumption requirements are the central key to a smooth automation of the tests covering topics like *when exiting this test before step*

10, which entry criteria does it comply to or which resumption sequence has to be executed to continue testing.

Walkthrough covering a step-by-step approach of the test plan.

Environmental requirements going into the details of what is needed concerning the test environment, e. g. tools, adapter, data preparation.

Discrepancy Reports identifying the defects.

Key Participants describing the assignment and task for each role involved.

Accreditation of Participants is a crucial point inside the open community since the EN 50128 requires e. g. the assessor to be outside of the project or organization. Thus the role of the assessor of an open community still has to be defined since per definition everybody can participate.

V&V Participants listing the partners participating in V&V activities,

Other participants including other interest groups such as reviewer by affiliate partners[7].

Timeline giving the timeline for the baselines as input to the V&V process and identifying when each artifact should be created.

To ease contribution from different parties and to reduce the testing effort, a high degree of formalization is targeted.

In realizing V&V, the selection, qualification and improvement of tools will be a central topic for research in the project. It will draw on outside results like the TOPCASED [Top] or the Project P [opea], and partners of the project will contribute some of there own tools to the openETCS pool for V&V. It will be considered to which extent the second criterion of "open proof", automatically verifiable proof, can be achieved with those means. There are mainly two ways to verify automated proving: Either by achieving trust in the prover, that is in effect verifying the tool, or by checking the evidence. The former is obviously very hard for tools which are continuously enhanced, which happens usually in open communities. And it will also be difficult for heavy weight tools like model checkers with high performance requirements which employ advanced algorithms and use sophisticated data structures.

Independently verifying the results seems more attractive. An example would be a proof engine like Coq [INR] which has a small "certification engine" to check proofs. Also approaches for model checkers [ZM03, Nam01] have been proposed, where the checkers produce evidence for verification (which might consist of huge amounts of data) on demand. An application scenario might use the checkers in the ordinary way for all usual development work, and turn on evidence generation and subsequent independent checking just for producing the safety case. Testing activities itself lend themselves much better to independent verification. For instance, one may measure test coverage with a small, validated tools. A demonstration of the viability of such approaches will have an important impact on the success of the openETCS project.

[7] affiliate partners are non-funded companies who signed the project cooperation agreement and with it get read access to the repositories starting from incubation phase to contribute e. g. by reviewing

3 Conclusion

Reconciling the idea of an open development with the strictly regulated world of safety-critical rail systems is an ambitious endeavor, and its concepts are not yet detailed. Yet in view of its potential, it is certainly worthwhile to try. Both from a conceptual as well as an economic point of view, making it real seems highly desirable. In this paper we sketched the approach taken by the openETCS project.

The next steps in contributing to its realization are detailing the verification and validation plan for the EVC software and selecting tools and methods, and also outlining what needs to be done to make them suitable for being used for developing part a system which in the end has to be certified.

Acknowledgment We acknowledge the comments of the anonymous referees which helped to improve the paper.

References

[Has12] Klaus-Rüdiger Hase. openETCS Workshop. Presentation, March 2012.

[INR] INRIA. http://inria.coq.fr/.

[ITE12] ITEA, 2012. http://www.itea2.org/project/index/view/?project=10135/.

[Nam01] K. S. Namjoshi. Certifying Model Checkers. In G. Berry, H.Comon, and A. Finkel, editors, *CAV 2001*, LNCS 2102, pages 02–13. Springer, 2001.

[opea] openDO. http://www.open-do.org/projects/p/.

[opeb] openETCS. https://github.com/openETCS.

[opec] openETCS. http://www.itea2.org/project/index/view/?project=10135/.

[oped] openETCS. http://openetcs.org/projects/.

[opee] openProofs. http://www.openproofs.org.

[Top] Topcased. http://www.topcased.org.

[Whe06] David A Wheeler. High Assurance (for Security or Safety) and Free-Libre / Open Source Software (FLOSS)... with Lots on Formal Methods / Software Verification. http://www.dwheeler.com/essays/high-assurance-floss.html, 2006. updated 2012.

[ZM03] L. Zhang and S. Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *DATE*, pages 10880–10885. IEEE, 2003.

Pattern-based methods for model-based safety-critical software architecture design: A PhD thesis proposal

Maged Khalil

Software and Systems Engineering
fortiss GmbH
Guerickestr. 25
80809 München
khalil@fortiss.org

Abstract: Software-intensive systems that perform safety-critical tasks are increasingly prevalent and pervasive in today's world. Driven by the incessant increase in the number of integrated control units, communication systems and software, managing architectural complexity, let alone mastering it, is becoming an increasingly difficult task.

Safety standards recommend (if not dictate) performing many analyses during the concept phase of development as well as the early adoption of multiple measures at the architectural design level, most of which has become part of the day-today business of safety-critical development, yet has to receive adequate tool support. This is particularly true if one wishes to front-load these aspects into an integrated solution environment, in which these (mostly) repetitive tasks can be automated. Model-based development techniques are increasingly used and well suited for these parameters.

Combining argumentation logic used for safety cases and safety concepts with abstract reasoning using model-based system description I argue about architectural optimization for safety-critical development. My approach allows reasoning about the system through the use of compositional description, which integrates physical environment models with system functional description models, and links problem-solving patterns with component model libraries which include nominal as well as faulty behavior.

1 Background and Motivation

1.1 Background

From advanced driver assistance and hybrid drives over autonomous high-speed rail and high precision medical equipment to Unmanned Aerial Vehicles: software-intensive systems that perform safety-critical tasks are increasingly prevalent and pervasive in today's world. Driven by the incessant increase in the number of integrated control units,

communication systems and software, managing architectural complexity, let alone mastering it, is becoming an increasingly difficult task. This difficulty in turn translates into a heightened possibility of design and implementation errors going undetected with hazardous consequences. This challenging situation is further exacerbated by the coupling of new development methods being employed for innovative technologies with increasingly complex international safety standards.

Non-functional requirements [La09] (the moniker under which safety is also bundled) should be dealt with from the beginning and throughout the software development process [CN00]. Ineffectively dealing with NFRs has led to a series of failures in software development [BLF99], [Li93]. Numerous literature examples have long existed [Br87], [CL99], pointing out these requirements as the most expensive and difficult to deal with. Functional requirements are naturally expressed in discrete, logic-based formalisms, which facilitate arguing about them at the architectural level. However, to express many non-functional requirements, software designers need to be able to integrate real-valued quantities representing physical constraints and probabilities into architectural description [HS06].

Safety standards recommend (if not dictate) performing many analyses during the concept phase of development as well as the early adoption of multiple measures at the architectural design level [EN08], [IEC09], [ISO11], [ARP11], [DO92], [DO12], most of which has become part of the day-to-day business of safety-critical development.

Because these analyses and architectural measures have to be conducted during the design phase of the systems, they cannot be based on physical prototypes and implemented software, and, hence, have to rely on the design information in terms of requirements, the structural design, functional specifications, and the principled knowledge about the (nominal and potential deviating) behavior of the components used [Bö11]. The complexity of such systems (regarding both structure and behavior) makes this a sophisticated task for experts and combined with the repetitive nature of the analyses carried out calls for computer-based automation and support. Any computer tool for this task that goes beyond editors and calculations and aims at supporting the core inferences, namely determining the consequences and risks implied by assumed faults based on design information and 1st principles knowledge, has to be knowledge-based, or, more specifically, model-based.

1.2 Related Work

Indeed, there exist results from previous work that form a good starting point for such an attempt. The field of model-based problem solving [St11] has generated theoretical foundations, prototypical solutions, and products for modeling artifacts and natural systems and for using models to solve a variety of tasks, with a major focus on automated diagnosis, but also solutions for failure modeling as well as for automated software FMEA [PS08], [SF12]. Several projects also exist which target the inclusion of reasoning about safety into the foundation of software development activities at the meta-modeling level in industry wide efforts, such as the AUTOSAR software reference architecture or EAST-ADL as well as the many research projects addressing these areas

(on the European scene alone, MAENAD, ATTEST2, TIMMO-2-USE, AMALTHEA, SAFE… etc.), which have shown progress but are far from a comprehensive solution.

On the other hand, there is extensive work on formalized reasoning about safety-cases and safety concept argumentation [PM99], [KE04], [KW04], which culminated in a standardization of the structured notation known as GSN [KH11]. An analysis of industrial safety-cases yielded a finite set of patterns [WS10] from which we plan to construct a pattern library of problem types and their respective solutions and use it to automate the generation of and argumentation about safety concepts as well as safety cases in our research CASE tool AutoFOCUS3 [AF3].

Several works exist on architectural benchmarking and design space exploration for architectural constraints. Works seeking to structure and facilitate the selection of architectural patterns and measures adequate for safety-critical requirements have already been presented [Ar10], but the selection process is still highly manual and case dependent.

1.3 Motivation

Yet there is no unified systematic approach to integrate safety-critical architectural constraints into the early phases of product development, before functions are divided unto logical or hardware components (such as all UML based approaches), while approaches that attempt to describe functionality free from component attachments are mostly limited to ADLs (e.g. feature models in EAST-ADL) and as such do not yet offer a seamless development capability. More specifically there is no approach that derives this integration from a solid argumentation about the selected solution pattern, its validity or the effects on software architecture correctness. Various approaches rooted in architectural design principles have tried to practically handle this discrepancy by recommending patterns which achieve many of the architectural constraints recommended by the standards, such as freedom from interference, usually by partitioning (separation of critical from non-critical software), and multiple redundancies (multiple version, dissimilar or independent software). Indeed, robust software architectures and efficient algorithms are still designed individually, not automatically generated, and this will likely remain so for some time. The emphasis, therefore, shifts from design synthesis to design verification—the proof of correctness. In summary, integrated models, methods and systems for fault analysis and risk assessment of cyber-physical systems are lacking, which is why I propose to develop coherent modeling formalisms and inference engines covering both physical and software systems, allowing the selection of and argumentation about architectural suitability and optimization and starting from the foundations stated above. Adequate tool support is central to the proposed approach, more so to deal with the highly repetitive nature of the complex tasks involved (e.g., FMEA), especially when dealing with variants, as most analyses have to repeated for every (even seemingly trivial) change, which would otherwise increase complexity and costs dramatically.

2 Solution Proposal

As previously mentioned, an approach based on compositional elements using which new or modified system variants can be (semi-)automatically (re-)configured and generated. More precisely; a model-based approach based on the consistent use of problem solving patterns and model element libraries.

2.1 Rationale

The proposed approach will be supported by a tool implementation based on an (deeper than hitherto achieved) integration of the physical system modeling and software component modeling which exploits the synergies between classical engineering, model-based problem solving and model-based software development paradigms. This approach is based on the following reasoning.

1- Hazard definition and avoidance is solely inferred by the interaction of the vehicle - as a physical system - with its physical environment (e.g., Vehicle under-braking results in an increased stopping distance which increases risk of collision.) A possible exception is an erroneous display of critical information to the driver, which leads to a critically wrong decision.

2- The behavioural model of the physical parts of the system and its interaction with the environment thus provides the primary inferences governing the requirements, analysis and function of the embedded software.

3- The embedded software thus only interacts with the physical system through this interface. Moreover, software errors do not directly lead to hazards in the environment per se, but rather must first lead to hazardous behaviour in the physical system by erroneous manipulation of the interface signals. As such, an incorrect vehicle speed display is not necessarily hazardous, while the severity of a false braking signal can only be determined by its impact on the physical system and its interaction with the environment. This view will govern the approach's handling of analysis for software components.

4- The functional and safety requirements for the embedded software are as such initially determined by the interaction model of the physical components with the environment. This model focuses the requirements and essential functions of the software: functional requirements are limited to the physically possible or relevant input and output combinations at the interface description, while the safety requirements are determined by the safety-critical scenarios and the physical actions leading to them.

2.2 Goals and Requirements

Thus a clear and concise understanding and representation of system functions, their relationships and dependencies as well as their their effect in the real world is essential. To manage complexity and because many (especially implementation) details are still unknown in early project phases; this representation must maintain a suitable level of abstraction, preferably at the functional network (functionality specification) level and before deployment to logical components. Furthermore, this representation must not only offer an abstract view of the system functionality during nominal component behaviour, but must support linking or deriving error models from them.

To master the complexity of current and future systems, the representation of complex operation modes and varying (partial-) functionality should be supported. This includes functional degradation and fall-back solutions..

By integrating hazard and risk analyses into the development environment, this approach provides a basis for the systematic and automated derivation of requirements, including relevance and completeness checks, for the safety concept and subsequent solutions as well as possibly enabling a synthesis of functional component description based on the physical models.

2.3 Summary

Underlying the approach is the consistent use of patterns from the categorization of hazard types, over the abstract modeling of the respective safety concepts, and down to their implementation in the system architecture description, with a focus on providing argument chains. The use of component model libraries that include with integrated behaviour descriptions for nominal and faulty conditions plays a central role. This could allow, for example, the automation of safety-critical optimized architecture pattern selection, based on the work in [Ar10] and in a later step, possibly a (semi-) automated architecture generation for each (sub-) system according to the selected safety concept.

This submission represents a work-in-progress, with several research project results still in various stages of completeness, and is meant to serve as a starting point for further discussion. A more detailed overview of the approach used and first results and prototypes are the subject of a future paper.

References

[AF3] AutoFOCUS 3, research CASE tool, af3.fortiss.org, 2012 fortiss GmbH

[ARP11] European Organisation for Civil Aviation Equipment, EUROCAE ED-79/ARP-4754: Certification considerations for highly integrated aircraft, 2011.

[Ar10] Ashraf Armoush, “Design Patterns for Safety-Critical Embedded Systems”, Ph.D. Thesis, RWTH-Aachen, 2010

[BLF99] K.K. Breitman, J.C.S.P. Leite, and A. Finkelstein, The World's Stage: A Survey on Requirements Engineering Using a Real-Life Case Study, J. of the Brazilian Computer Soc., vol. 6, no. 1, pp. 13-38, July 1999.

[Bö11] „Funktionale Sicherheit: Grundzüge sicherheitstechnischer Systeme", Börcsök, J. / VDE-Verlag, 2011

[Br87] F.P. Brooks Jr., No Silver Bullet: Essences and Accidents of Software Engineering, Computer, no. 4, pp. 10-19, Apr. 1987.

[CL99] L.M. Cysneiros and J.C.S.P. Leite, Integrating Non-Functional Requirements into Data Model, Proc. Fourth Int'l Symp. Requirements Eng., June 1999.

[CNY00] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, Non-Functional Requirements in Software Engineering. Kluwer Academic, 2000.

[DO92] European Organization for Civil Aviation Equipment (EUROCAE) and United States Department of Defense (DOD), EUROCAE ED-12B/DO-178B: Software considerations in airborne systems and equipment certification, 1992.

[DO12] United States Department of Defense (DOD), DO-178C: Software considerations in airborne systems and equipment certification, 2012.

[EN08] European Committee for ElectroTechnical Standardization, EN-50126 Railway Standard, www.cenelec.eu, 2008.

[GH08] Grunske, L. and Jun Han, A Comparative Study into Architecture-Based Safety Evaluation Methodologies Using AADL's Error Annex and Failure Propagation Models, HASE 2008. 11th IEEE High Assurance Systems Engineering Symposium, 2008.

[HS06] Thomas A. Henzinger, EPFL & Joseph Sifakis, Verimag, The Discipline of Embedded Systems Design, IEEE Computer Society 2007 – an update of The Embedded Systems Design Challenge, Proc. 14th Int'l Symp. Formal Methods, LNCS 4085, Springer, 2006, pp. 1-15.

[IEC09] International Electrotechnical Commission, IEC 61508 Standard, "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems", www.iec.ch/functionalsafety, 2009.

[ISO11] International Standards Organization, ISO 26262 Standard, Road Vehicles Functional Safety, www.iso.org, 2011.

[Ke04] Tim Kelly, A Systematic Approach to Safety Case Management, SAE 2004 World Congress & Exhibition, March 2004, Detroit, MI, USA, Session: Safety-Critical Systems.

[KH11] Tim Kelly, Ibrahim Habli, et al.: Goal Structuring Notation (GSN). GSN COMMUNITY STANDARD VERSION 1, Origin Consulting (York) Limited, on behalf of the Contributors, November 2011

[KW04] Tim Kelly, Rob Weaver, The Goal Structuring Notation – A Safety Argument Notation, Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases.

[La09] Axel van Lamsweerde, Requirements Engineering - From System Goals to UML Models to Software Specifications, 2009.

[Li93] D.R. Lindstrom, Five Ways to Destroy a Development Project, IEEE Software, pp. 55-58, Sept. 1993.

[PM99] Papadopoulos Y., McDermid J.A., The Potential for a Generic Approach to Certification of Safety-Critical Systems in the Transportation Sector, Reliability Engineering and System Safety, 63(1):47-66, Elsevier Science,1999.

[PS08] Chris Price and Neal Snooke, An Automated Software FMEA, Proceedings of the International System Safety Regional Conference, Singapore, April 2008

[SF12] Struss, P., Fraracci, A.: Automated Model-based FMEA of a Braking System. In: 8th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes (Safeprocess 2012), Mexico City, 2012

[St11] Struss, P. : A Conceptualization and General Architecture of Intelligent Decision Support Systems. MODSIM2011, 19th International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2011, pp. 2282-2288. ISBN: 978-0-9872143-1-7.

[WS10] Stefan Wagner, Bernhard Schätz, Stefan Puchner, Peter Kock, A Case Study on Safety Cases in the Automotive Domain: Modules, Patterns, and Models, Proc. International Symposium on Software Reliability Engineering (ISSRE '10), IEEE Computer Society, 2010

Integrating State Machine Analysis with System-Theoretic Process Analysis

Asim Abdulkhaleq, Stefan Wagner

Institute of Software Technology
University of Stuttgart, Germany
asim.abdulkhaleq@informatik.uni-stuttgart.de
stefan.wagner@informatik.uni-stuttgart.de

Abstract: Safety becomes a critical aspect for software-intensive systems in different applications areas. Many hazard analysis techniques are proposed and used to investigate system design models to elicit hazards and design flaws. STPA (System-Theoretic Process Analysis) is a modern hazard analysis technique, which is based on a new systems-theoretic model of accidents for large and complex systems. With STPA, the system is viewed as interacting control loops and the accidents are considered as results from inadequate enforcement of safety constraints in design, development and operation. STPA still needs appropriate diagrammatic notations to represent the relation between the process model variables, control actions and hazards. For this purpose, we propose to integrate state machine analysis with STPA to provide a suitable notation of arguments between the states of controllers, control actions and hazards.

1 Introduction

A great challenge today in the development of software-intensive systems is how to develop a safe system that fulfills safety requirements and ensures safe functions of a system under all safety conditions. Thus, the safety analysis of a system needs to provide an understanding about the possibility of accidents occurring in the system. Safety analysis supports examining and investigating systems or subsystems to identify and classify each potential hazard according to its severity and likelihood of occurrences. It is also extremely important to prevent the hazards which can lead to injury and even loss of life. Increasing complexity and size of modern systems makes system safety a great challenge on how to design such system with acceptable level of risk.

A number of hazard analysis techniques have been proposed to perform system hazard analysis. Traditional hazard analysis techniques such as Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA)[Eri05] address only the failure of individual components and view the accidents as resulting from a chain or sequence of events. Instead, STPA analyses the complete system and addresses the component interaction failures and component failures as well. STPA is an efficient hazard analysis technique which has been developed by Leveson based on system theory [Lev12]. The main purpose of

STPA is to identify the new causal factors such as design errors, including software flaws, component interaction accidents, cognitively complex human decision-making error and social, organisational and management factors contributing to accidents that are not addressed by traditional hazard analysis techniques[Lev12, Tho12, Lev04].

Indeed, the hazard analysis techniques cannot describe well the dynamic behaviour and state transition of complex system. State machine models have emerged as one important modelling perspective on the reactive behaviour of complex systems. They represent the complete behaviour of the system using states and transitions and can easily define the constraints on transitions.

Problem Statement The safety analyst during STPA must augment the control structure with process models and examine the controller with process model variables to see if this process path can lead to unsafe control actions or not. STPA does not show, however, how to make these arguments.

Research Objectives The overall objective of this research is to fill this gap and find ways for including and better analysing the dynamic behaviour of systems during STPA hazard analysis. We plan to investigate various modelling and analysis techniques. In this paper, we focus on proposing the integration with state machine models.

Contribution For that, we integrated the state machine analysis with STPA to show how this formal modelling technique can be used to support the safety analysis practices carried out with STPA. The resultant methodology can assist safety analysts during STPA step three to investigate inadequate control actions and verify each path in the control loop with process state variables. Furthermore, we applied the proposed methodology to Anti-lock Braking System [Gmb07] to explore its advantages and limitations.

2 Background

2.1 STPA Hazard Analysis

STPA is a top down analysis approach that considers the dysfunctional interactions between software, hardware, operators, management and regulatory bodies. We summarise the main steps of STPA below [Lev12]:

1. **Step 1:** Establish the fundamentals of STPA before beginning the analysis by identifying system accidents or unacceptable loss events. Next, identify the hazards for the system and translate them into top-level system safety constraints. Next, draw a preliminary (high-level) safety control structure which depicts the components of the system and the paths of control and feedback.

2. **Step 2:** Use the control structure defined in step 1 as guide for investigating the analysis; identify the potentially unsafe control actions that could lead to a hazardous state. Then, refine system safety constraints according to these unsafe control actions. STPA uses control structure diagrams and system hazards to generate the system and component safety constraints and safety requirements.

3. **Step 3:** Determine how each potentially hazardous control action, which was identified in step 1, could occur. Next, identify the process models of the controllers. Finally, augment the control structure with process models for each control component and examine the paths of the control loop to see if they could cause unsafe control action. Recommendation for the system design should be developed for additional mitigations.

2.2 State Machine Analysis

Finite State Machines (FSM) are a formal model for describing the dynamic behaviour of a system by using states and state transitions. It shows how the input drives changes in the state of a system and how output is produced [NRJA11]. It consists of a set of input events, a set of states, an initial state and a set of transitions.

Finite state machines are known as event-driven models which have been widely used in different areas for representing discrete events of systems. They can be used to model both functional and dysfunctional behaviours of system. FSM can be used also to analyze the trace of the accidents and it takes the occurrence of system failure as trigger event where the accidents arose as the result of inadequate enforcement of constraints on system behaviour [FGZZ11, AP11]. Therefore, using the characteristics of the event driven state transition of FSM in the safety analysis can aid effectively in providing a way to notate and assess the process model paths of system and whether they lead to hazards.

3 Proposed Methodology

As stated in the previous section, STPA shows the system behaviour as a set of interconnected boxes that represent the sub-components of system model and their relations. STPA addresses component interaction failures and component failures. FSM can model the dynamic behaviour of components and complete systems and, hence, enable the assessment of both. According to our experience with STPA hazard analysis, we found that the integration between FSM and STPA could assist the safety analyst in identifying and evaluating dysfunctional behaviour of a system. The goal of analysing the behaviour of a system is to identify the hazardous control actions by considering the operating modes of the system and mitigating the catastrophic effects. Our proposed methodology (as shown in Figure 1) aims to:

- Use STPA to identify the potential for inadequate scenarios of the system that could

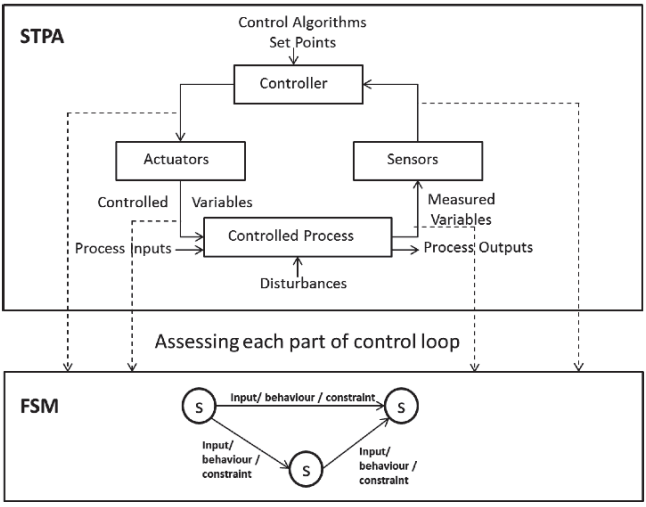


Figure 1: The integrating of FSM with STPA.

lead to a hazardous state.

- Identify the process model of each controller and identify its process state variables.
- Use FSM to model the dynamic behaviour of the controllers and show how the behaviour changes over the history of its inputs.
- Use FSM to model the safety constraints identified using STPA with states and state transitions of the system.
- Assess each part of the control loop of the system with FSM and analyse each part for identifying the hazardous control actions based on all the possible states, which have an effect on the control action.
- Extend the control action table, identified using STPA in step 2, to include the operating modes (system states) and its effect on the control action as additional columns.
- Refine the safety constraints and design decisions.

4 The Integration of FSM with STPA

An illustrative example is given here, which considers an Anti-lock Braking System (ABS) for modern automobile, to show the hazard analysis with STPA and how to integrate FSM with STPA. First, we applied STPA to ABS to identify unsafe control actions and causal factors. Then, driven by the STPA results, we mapped these results to an FSM. Finally, we refined the control action table and the safety constraints.

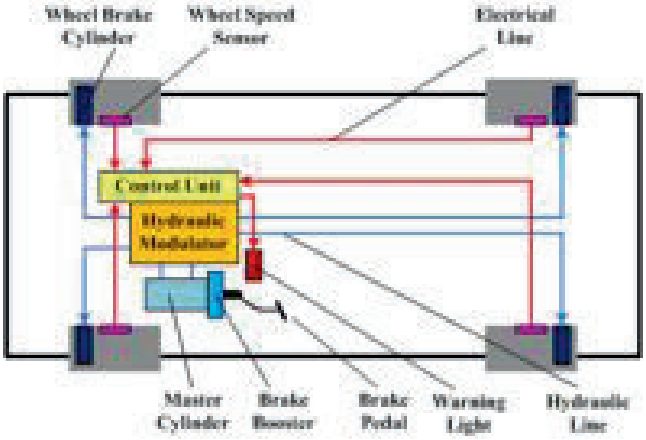


Figure 2: The ABS architecture [Cle12].

4.1 An Illustrative Example: Anti-Lock Braking System (ABS)

ABS is now becoming a standard in the automotive world and it assists the driver by preventing the wheels from completely locking during an emergency stop by applying the optimum braking pressure to the individual wheels, thus ensuring the vehicle can still be steered and shortening braking distances on slippery surfaces [Cle12, AZHS11, Gmb07].

Typical ABS components (as shown in Figure 2) include[Cle12, AZHS11]: *Electronic Control Unit (ECU)*, which acts as controller unit and receives information from the sensors, determines when a wheel is about to lockup and controls the hydraulic control unit;*Hydraulic Control Unit (HCU)*, which is also called hydraulic pump and controls the pressure in the brake lines of the vehicle; *Modulator valves*, which are presented in the brake line of each break and are controlled by the hydraulic control unit to regulate the pressure in the brake lines; and *Wheel speed sensors* (up to 4), which measure wheel-speed and transmit information to an electronic control unit.

4.2 Hazard Analysis of the ABS

In accordance with the proposed methodology, we start the hazard analysis process with STPA based on the high-level control structure model of the system. Our analysis focuses specifically on the hazards that arise due to the interaction between electronic control unit and hydraulic control unit. For example, the ECU reads signals from the electronic sensors which monitor wheel rotation. If a wheel's rate of rotation suddenly decreases, the ECU orders HCU to reduce the line of pressure to that wheel's brake. In the following, we will describe in details the STPA hazard analysis process and the integration of FSM with it:

Step 1: *The safety analyst must establish the following fundamentals:*

- **System Level Goals:**
  - **G.1:** Prevent wheel lockup during an emergency stop.
  - **G.2:** Provide controlled stopping by maintaining maximum tire to road friction.
  - **G.3:** Permit the driver to maintain steering while braking.
- **Accident:** The accident to be considered is the ABS vehicle crashes with a vehicle and the occupants are injured while ABS is involved (A.1).
- **System Level Hazards:**
  - **H.1:** Loss of steering control during braking operation (A.1).
  - **H.2:** ABS did not manipulate optimal wheel slip and stop in the shortest distance (A.1).
- **Safety Constraints:**
  - ABS must engage automatically when rapid deceleration are detected.
  - ABS must sense the motion of each wheel to detect a skid condition and be able to pulse the braking hundreds of times per second.
- **Design Constraints:**
  - ABS must be engaged after brake pedal is pressed.
  - ABS must prevent wheel lock-up in hard braking situations (e.g. when the speed of vehicle is over 15 mph) when lockup may occur.
- **Design Requirements:**
  - ABS shall stop the vehicle in the shortest distance.
  - ABS shall maintain the safe stop distance on loose road surface (e.g. snow-covered, gravel-roads, etc.).
- **Functional Control Structure:** Once the hazards to be evaluated have been reviewed, the safety analyst develops a control structure diagram of the system. Figure 3 shows the control structure diagram which depicts the interacting control loop between ECU and HCU in ABS system.

Step 2: *Identify potentially unsafe control actions*: Based on the control structure in Figure 3, we can identify the potentially unsafe control actions, which can lead to a hazardous state. A hazardous state is a state that violates the safety constraints that are defined for the system. Each control action should be analysed in four ways (shown in Table 1) to determine if it is hazardous as defined by the system-level hazards or not. For example, if we consider the situation that the brake pedal is pressed during an emergency stopping situation in which wheels lock up but the ABS does not engage. This situation can be considered as hazardous because it can lead to hazard H.1 in our example.

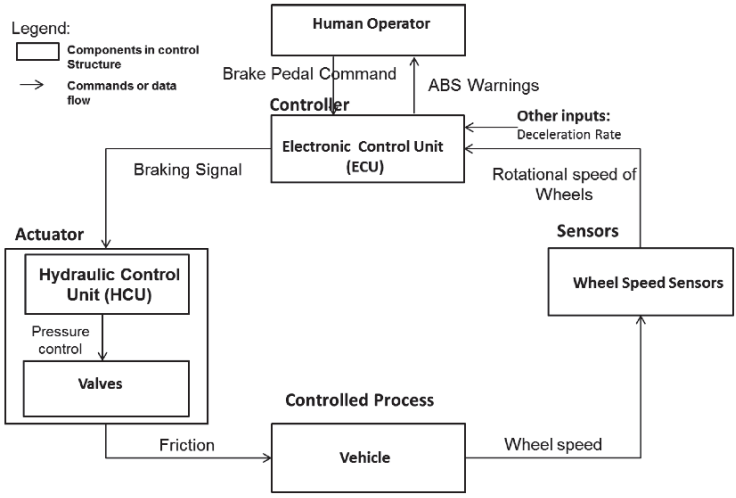


Figure 3: The control structure of ABS (ECU-HCU).

Step 3: *Causal factor analysis*: This step of STPA is first to augment the control structure (Figure 2) with process models and then to determine how hazardous control actions (Table 1) could occur. Figure 4 shows the process models for the ECU and human operator as an example. In this process model, the ECU senses the brake pedal's situation and receives the information about the wheel rotational speed from speed sensors. If it detects wheel locking, it reduces the braking force repeatedly until the wheel starts rotating again. Based on the process models developed, the next step is to identify the causal factors for the hazards. The causal analysis starts with each hazardous control action identified in Step 2 to determine how it could happen. In the control structure diagram of ABS, an unsafe behaviour can be resulted from either a missing or an inadequate constraint on the process or inadequate enforcement of the constraint that leads to its violation. As an example, consider the unsafe control action of not braking when the brake pedal is pressed. In this case, the hazard in the controller component could result if the information from the wheel sensors or the brake pedal command is not provided or is not implemented correctly or the process model is incorrect. Consequently, there are three cases which can be considered: 1) the brake pedal command is sent but not received by the ECU; 2) the ECU received the brake pedal command but it does not execute it. After the causal factor analysis is done, more details can be performed and 3) the brake pedal command is not sent to the ECU (component is defect).

4.3 FSM Construction

So far, there is no diagrammatic notation to represent the relation between the process model state variables (system states), control actions and safety constraints. Moreover,

Table 1: Examples of potentially inadequate control actions of the ABS system

Control Action	Not Given	Given Incorrectly	Wrong Timing or Order Cause Hazard	Stopped too soon or applied too long
Brake pedal command	Brake event applied but not received by ABS [see H.1]	Brake event is too short	Brake provided too late	Brake stopped too soon
Wheel speed sensors	Wheel speed data does not provide [see H.1, H.2]	Wrong current wheel speed provided [see H.1]	Current wheel speed updated too late [see H.1]	Wheel speed sensors stopped too soon [see H.2]

STPA does not include the operating modes of components, which will have an effect on the causal factors analysis and may determine the safety of the action or event. Therefore, FSM can be used to determine the system states that affect the safety of the control actions. In our example, the ECU controller has four operating modes: *inactive*, *handelLock*, *applyBrakePedal* and *reducePressure*. For the valve actuator component, there are three modes: *open*, *block* and *release* and for HCU actuator, there are three modes: *inactive*, *stopPump* and *openPump*. Next, we model the dynamic behaviour of the system by constructing a finite state machine which visualises the system states, control actions and safety constraints to support the safety analyst during STPA step three. Figure 5 shows the finite state machine of the controller in the ABS system. By using FSM, we are able to show the relations of potential combinations of relevant process state variables and according to these states we can identify the control action and determine whether issuing that control action leads to a hazardous state. As an example, if we consider the control action *brake pedal command* that can be a hazardous control action, it consists of the values of the following process model state variables: the brake pedal is pressed, the deceleration rate exceeds a preset maximum level, valve is close, wheel is locked and hydraulic pressure is reduced. Table 2 specifies the modified control action table for the brake pedal command based on our proposed methodology as an example.

Each row in the table 2 should be evaluated to determine whether it is hazardous as defined by the system-level hazards. We can notice that, there are some combinations which cannot be hazardous; therefore it should be neglected from the table. This process will continue to combine other potential states which are related to specified control action. Consequently, the safety constraints will be refined and documented.

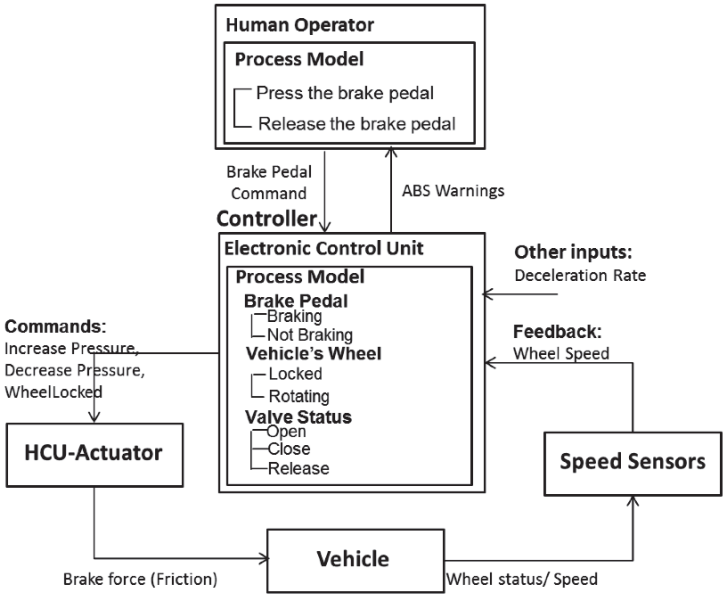


Figure 4: Control structure of the ABS with the process model of the electronic control unit

5 Discussion

To determine the usefulness of our proposed methodology to support safety analyst during STPA, we compared the results which are achieved by STPA (Table 1) with our results in Table 2. As we can see in Table 1, each control action has been analysed in four ways, which consider only the timing information of the control action (i.e. not given, too short, too late, too early, out of sequence-wrong timing). There is no systematic way to let the safety analyst know how to evaluate each control action and determine which can lead to a hazardous state and it is not clear how he or she can perform it. That means the safety analyst's professional knowledge and experience play a critical role in this step to evaluate the control actions. In most cases, the control actions in the control loop can depend on a number of relevant factors such as system states, human behaviour and environment conditions which have effects on determining the safety of the control action. Therefore, it is important to have a modelling method that allows considering all relevant factors. The FSM can visualise the actual behaviour of the system and can be used to examine the hazardous behaviour. We used FSM to examine each control action in the loop to detect hazardous situations based on the potential combinations of system states that are relevant. It is important to note that we do not consider the human behaviour and environment conditions (e.g. road surface) in our example, we only consider the system operating modes (states) that are relevant to control actions.

Many systems may exhibit behaviour much more complex. Thereby, in the evaluation process of control actions, it will be hard to determine unsafe control actions and it may

Table 2: The control action table for the *brake pedal command* based on the potential combination of system states

Control Action	Wheel Status	Wheel Speed	Valve Status	Hazardous?
Brake pedal command	Locked	slow	open	Yes
Brake pedal command	Locked	fast	open	Yes
Brake pedal command	Locked	slow	close	No
Brake pedal command	Locked	fast	close	No
Brake pedal command	rotating	slow	open	Yes
Brake pedal command	rotating	fast	open	Yes
Brake pedal command	rotating	slow	close	No
Brake pedal command	rotating	fast	close	No

take more time and effort. Moreover, the control loop of the system may have one or more controllers which control one process. Therefore, it is possible to find the different kind of control actions in the control loop such as conflict control actions or overload control actions or dependent control actions. In this case, the safety of such control actions will rely on the multiple controllers' behaviours. We found STPA does not provide a description on how to detect the hazardous situations among multiple controllers.

Consequently, the benefits which can be gained by integrating FSM during STPA are, first to model the complex system behaviour. Even though FSM of complex system will become more complex, it provides an understandable notation of dynamic behaviour and states of complex systems. A second benefit is to prove the consistency, correctness and completeness of the STPA analysis based on FSM by using model checking [EK00, DAC99, TS03]. Model checking is an efficient verification technique for models expressed as finite state machines. Model checking can help the safety analyst to automatically check if the FSM of system has the desired properties. We believe that model checking can be helpful in evaluating control actions based on our proposed methodology.

According to our proposed methodology, the safety analyst have to determine the possibilities combinations of states which are related to the control action and investigate how the system behaves in presence of different types of hazardous control actions. For this purpose, the safety analyst will use FSM to determine these relevant combinations. In fact, formal methods, such as FSM, are accepted best, if verification can be automated. Model checkers can do this for the finite state machines. For these reasons, we plan as future work to use model checkers in our proposed methodology for proving the corresponding correctness properties of finite-state machines and use existing commercial modelling tool such as Simulink [DH04] for simulating and analysing multidomain dynamic systems. Finally, in this paper, we identified important research challenges that need to be resolved to

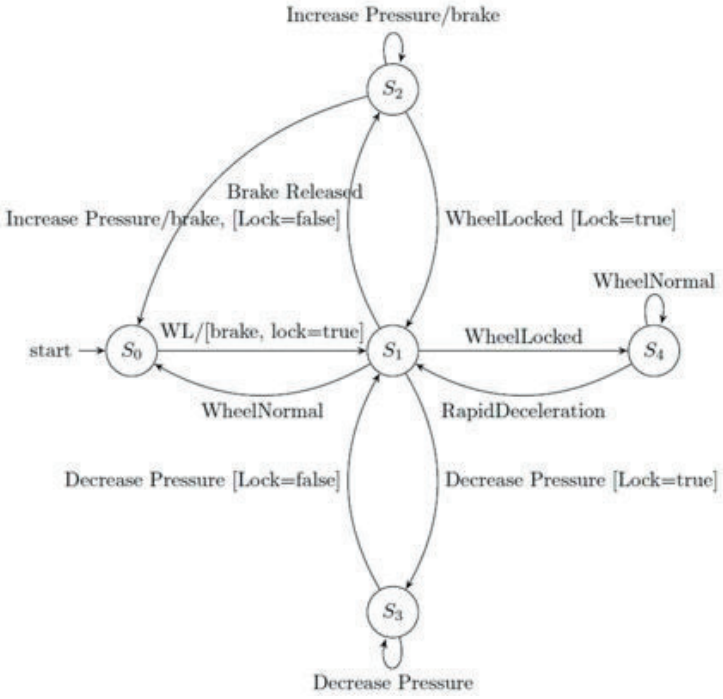


Figure 5: Finite state machine of ECU controller in the ABS system.
Where: *S0 = inactive (brake not pressed), S1=handellock, S2= applyBreakPedal, S3= pressureReduction, S4= MonitorDeceleration and WL= WheelLocked.*

make the hazard analysis with STPA practical.

6 Related Work

There a large body of existing work about integrating formal methods with hazard analysis techniques. Here we discuss some of the most closely related methodologies and tools which are used.

Thomas [Tho12] presents a formal mathematical structure underlying STPA and describes a procedure for systematically performing an STPA analysis based on the formal structure. He also presents a method for using the result of the hazard analysis to generate formal safety-critical, model-based system and software requirements. He points out that he is working on exploring potential ways in which similar kinds of detailed procedures can be created to assist the safety analyst during STPA step two.

Ariss et al. [EAXW11], present an approach for integrating fault-tree-based safety analysis

into statechart-based functional modelling. Their resultant model from the integration shows how the system behaves when a failure condition occurs and acts as the basis model to ensure safety through requirements validations.

The work of Fan et al. [FGZZ11] is closely related to the topic of safety analysis for complex systems based on the finite state machine theory. They introduced a formal modelling method based on finite state machines and proposed a safety analysis and assessment method of complex system based formal model. They did not show how to use finite state machine with any hazard analysis techniques.

Using formal methods during safety analysis would be a useful way in which the dynamic behaviour of complex system is presented. Therefore, our proposed solution aims to integrate FSM during the third step of STPA to visualise the relations between system states and control actions identified by STPA to help a safety analyst to examine the control actions based on system states whether it can lead to hazardous state.

More past work has been a large amount of work in the integration of FSM with traditional hazards analysis techniques such as FTA and FMEA which address only the component failure. Our proposed methodology is based on STPA which creates a set of scenarios that can lead to a hazard, is the same as FTA but STPA includes a set of potential scenarios in which no failures occur but the problems arise due to unsafe and unintended interactions among the system components.

7 Conclusion

In this paper, a solution is proposed to assist in evaluating the control actions identified during STPA step two based on the system states using formal method. We integrated state machine analysis with STPA to provide a suitable notation of arguments between states of system and control actions. The proposed solution uses the result of STPA to notate the relations between system states and control actions. We selected the anti-lock braking system as an illustrative example. By applying STPA to ABS, we determined the control actions, process model and state variables of our example. Next, we constructed an FSM of the controller with state variables and examined each potentially hazardous control action by examining each potential combination of relevant system states. These combinations are related to the system-level hazards identified by STPA step one. Finally, the control action table as well as safety constraints are refined and updated. The scalability of our methodology largely depends on the scalability of STPA and finite state machine. STPA has been successfully applied to different systems with a wide range of complexity in different areas such as Space Shuttle Operations [OHD+08] and the Darlington Shutdown System [Son12]. The finite state machines also have been widely used in different areas for representing the dynamic behaviour of systems.

7.1 Limitations

There are two main limitations of our methodology. The first is getting an appropriate finite state machine of real system. We have constructed the finite state machine of our example from various published sources. The second is a larger number of states of complex system that may forces the safety analyst to devote much of his effort and time to construct the potential combinations of relevant states to control action.

7.2 Future work

As future work, we aim to extend the scope and depth of analysis with the proposed methodology and try to identify systematic procedures for it. Future work is needed to further refine this method by applying it to other cases. Therefore, we aim to apply this methodology to both adaptive cruise control and anti-lock braking system which work together in a vehicle and investigate their interactions, control actions and causal factors as a single study object. In addition, we will work on comprehensive tool support which will be the starting point for the proposed methodology to be automated. Furthermore, we also aim to investigate applicability of STPA to security problems to identify and mitigate the threats that could emerge in systems.

References

[AP11] V.S. Alagar and K. Periyasamy. *Specification of Software Systems*. Texts in Computer Science. Springer, 2011.

[AZHS11] A. A. Aly, E. Zeidan, A. Hamed, and F. Salem. An Antilock-Braking Systems (ABS) Control: A Technical Review, Intelligent Control and Automation. In *An Antilock-Braking Systems (ABS) Control: A Technical Review, Intelligent Control and Automation*. Intelligent Control and Automation, August 2011.

[Cle12] Anti-lock Braking Systems. The Clemson University Vehicular Electronics Laboratory, 2012.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[DH04] J. Dabney and T.L. Harman. *Mastering Simulink*. Pearson/Prentice Hall, 2004.

[EAXW11] O. El Ariss, Dianxiang Xu, and W.E. Wong. Integrating Safety Analysis With Functional Modeling. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 41(4):610 –624, july 2011.

[EK00] G. Eleftherakis and P. Kefalas. Towards Model Checking of Finite State Machines Extended with Memory through Refinement. In *Towards Model Checking of Finite State Machines Extended with Memory through Refinement*. Springer-Verlag, 2000.

[Eri05] Clifton A. Ericson. *Hazard Analysis Techniques for System Safety*. Wiley, 2005.

[FGZZ11] Yichen Fan, Qi Gong, Jianguo Zhang, and Yuanzhen Zhu. Safety analysis for complex system based on the finite state machine theory. In *Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on*, pages 594 –598, june 2011.

[Gmb07] Robert Bosch Gmbh. *Bosch Automotive Handbook*. Bosch Handbooks series. Bentley Pub, 2007.

[Lev04] Nancy G. Leveson. A Systems-Theoretic Approach to Safety in Software-Intensive Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):66–86, 2004.

[Lev12] N.G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. Engineering Systems. Mit Press, 2012.

[NRJA11] P.D. Nrusingh, D. Ranjan, C. Jayakar, and H. Arindam. Event Driven Programming for Embedded Systems- A Finite State Machine Based Approach. In *Event Driven Programming for Embedded Systems- A Finite State Machine Based Approach*. In: Proc. 6th International Conference on Systems, 2011.

[OHD+08] B.D. Owens, M.S. Herring, N. Dulac, N.G. Leveson, M.D. Ingham, and K.A. Weiss. Application of a Safety-Driven Design Methodology to an Outer Planet Exploration Mission. In *Aerospace Conference, 2008 IEEE*, pages 1 –24, march 2008.

[Son12] Y. Song. Applying System-Theoretic Accident Model and Process (STAMP) to Hazard Analysis. In *Applying System-Theoretic Accident Model and Process (STAMP) to Hazard Analysis*. McMaster University, 2012.

[Tho12] J. Thomas. Extending and Automating a Systems-Theoretic Hazard Analysis for Requirements Generation and Analysis. In *Extending and Automating a Systems-Theoretic Hazard Analysis for Requirements Generation and Analysis*. SAND2012-4080,Sandia National Labroratories., 2012.

[TS03] A. Thums and G. Schellhorn. Model checking FTA. In *FME 2003: Formal Methods, LNCS 2805*, pages 739–757. Springer-Verlag, 2003.

Zur Risikobestimmung bei Security-Analysen in der Eisenbahnsignaltechnik

Sebastian Saal[1] Dennis Klar[2] Markus Seemann[3] Michaela Huhn[2]
[1] Institut für Theoretische Informatik, Technische Universität Braunschweig
[2] Institut für Informatik, Technische Universität Clausthal
[3] IC MOL RA R&D, Siemens AG

Abstract: Die CORAS-Methode [LSS11] unterstützt eine systematische, defensive Risikoanalyse, die insbesondere zur Ermittlung von IT-Security-Anforderungen eingesetzt wird. Sie ermöglicht eine Risikobeurteilung auf Basis einer Systemanalyse und der Identifikation der zu schützenden Werte. Dafür wird ein System schrittweise in sogenannten Bedrohungsdiagrammen auf Schwachstellen untersucht. Danach erfolgt die Risikobewertung auf Grundlage des möglichen Schadenausmaßes und der zu erwartenden Häufigkeit von Angriffen.

In einer Fallstudie zur Risikoanalyse von IT-Security-Bedrohungen in sicherheitsrelevanten Systemen haben wir festgestellt, dass sich das Schadenausmaß durch die genaue Identifikation der Schutzziele mit der CORAS-Methode gut ermitteln lässt, während sich die Bewertung der Angriffshäufigkeit als schwierig erweist. Wir schlagen daher vor, die Angriffshäufigkeit durch eine semi-quantitative Klassifikation in mehreren Aspekten zu bewerten. In diesem Beitrag beschreiben wir eine derartige Bewertung, die die Aspekte *Motivation, Mittel und Gelegenheit* berücksichtigt, und diskutieren verschiedene Funktionen zur Kombination der Klassifikationen. Anhand der Fallstudie aus der Eisenbahnsignaltechnik zeigen wir, dass aus dem abgeleiteten Klassifikationschema eine sinnvolle Risikobewertung von IT-Security-Bedrohungen für sicherheitskritische Systeme resultiert.

1 Einleitung

In der Eisenbahnsignaltechnik rückt die IT-Sicherheit zunehmend in den Fokus der Entwicklung. Durch die intensive Vernetzung und neuartige Dienste der Komponenten entstehen neben den zahlreichen Vorteilen, wie neue Funktionen, erhöhte Verfügbarkeit und geringere Kosten, insbesondere auch neue Risiken, die analysiert und behandelt werden müssen. Zusätzliche Schnittstellen, z. B. für die Fernwartung, und die Nutzung offener Kommunikationsnetze (Funkverfahren, öffentliche Netze usw.) bergen neue Möglichkeiten für IT-Security-Angriffe, die auch die funktionale Sicherheit beeinträchtigen können. Daher werden neue Ansätze für entwicklungsbegleitende Risikoanalysen bezüglich der (IT-) Security benötigt.

Der Rahmen für die Security-Analyse wird durch die bahnspezifischen Risikostandards und -normen EN 50129 [CEN03], EN 50159 [CEN10] und den Entwurf eines Schutzprofils in der VDE 0831-102 [VDE13] vorgegeben. Konzepte und Methoden bleiben jedoch domänenspezifisch, so dass viele verschiedene Verfahren mit ihren individuellen Vorteilen

und Nachteilen zum Einsatz kommen. Ein Beispiel sind diverse tabellarische Ansätze, die aber bei größeren Systemen schnell an Übersichtlichkeit verlieren.

Hier wollen wir CORAS [LSS11] einsetzen, eine universellen, modellbasierten Ansatz zur Riskoanalyse. CORAS bietet zwei entscheidende Vorteile: zum Einen eine strikte Methodik mit acht Schritten und einem Leitfaden zur Bearbeitung. Zum Anderen eine graphische Analyse mit spezialisierten Diagrammtypen und Symbolen, die vergleichsweise übersichtlich und leicht verständlich sind. Insbesondere gegenüber tabellarischen Ansätzen werden so die Einflüsse und Wechselwirkungen besser verdeutlicht.

Eine generelle Schwierigkeit bei Risikoanalysen zeigt sich beim Übergang von der informellen Analyse zu einer quantitativen Auswertung. Für die analysierten Ereignisse müssen Eintrittswahrscheinlichkeiten oder -häufigkeiten abgeschätzt werden. Eine direkte Abschätzung ist jedoch, wie auch die CORAS-Autoren feststellen, aus verschiedenen Gründen häufig nicht möglich (vgl. [LSS11] S. 207f.), etwa wenn es keine Erfahrungswerte gibt, weil das System neu oder substantiell verändert ist oder die Einsatzbedingungen abweichen. Ebenso ist es schwierig, wenn ein unerwünschtes Ereignis sehr selten auftritt, gravierende Konsequenzen hat oder sein Eintreten nicht (zuverlässig) detektierbar ist.

Ein weiterer kritischer Punkt im Bereich IT-Security ist der Umgang mit den „technisch-abstrakten“ Größen der Wahrscheinlichkeit oder Häufigkeit an sich. Anders als bei Safety-Analysen müssen menschlich-psychologische Aspekte einfließen. Die Bandbreite einer möglichen Motivation hinter Security-Angriffen ist groß; teils sind sie von rationalen Kosten-Nutzen-Abwägungen geprägt (z. B. Wirtschaftskriminalität), teils extern motiviert oder gar irrational (z. B. „Spieltrieb“ bis hin zu Terrorismus).

Daher verfolgen wir in diesem Beitrag einen Ansatz, der *Einflussfaktoren* wie etwa die Motivation, die Gelegenheit oder die notwendigen Mittel für einen Angriff explizit in die Abschätzung von Security-Risiken – auch in der Systementwicklung – einbezieht. Die Wahrscheinlichkeiten werden aus verschiedenen Einflussfaktoren abgeleitet, die separat qualitativ abgeschätzt werden, z. B. durch begriffliche Einordnung in eine von mehreren Stufen (Levels). Anschließend erfolgt die Verrechnung der Einflussfaktoren mit einer geeigneten Gewichtung zu einem Häufigkeitslevel.

Ziel dieses Beitrags ist es darüber hinaus, die Anwendbarkeit von CORAS auf Systeme der Eisenbahnsignaltechnik zu demonstrieren. Dabei soll, im Hinblick auf die typische Anwendung bei Neuentwicklungen, ein auf „*Human Factors*“ basierender Ansatz in die quantitative Risikoanalyse von CORAS integriert werden. In Abschnitt 2 wird zunächst die CORAS-Methode vorgestellt. Abschnitt 3 erläutert die Anwendung von CORAS und die benötigten Erweiterungen. In den Abschnitten 4 und 5 folgt eine Fallstudie und eine Diskussion der Ergebnisse.

2 Risiko-Analyse nach CORAS

Die CORAS-Methode [LSS11] wurde entwickelt, um die Analyse von IT-Security-Risiken und -Bedrohungen zu erleichtern. Die Schwerpunkte sind einerseits einfache Kommunikation und Dokumentation, andererseits eine Formalisierung der Analyse.

2.1 Der CORAS-Ansatz

CORAS orientiert sich an den zu schützenden Werten (Assets), ist defensiv[1] ausgerichtet und implementiert den ISO 31000 Standard [Int09] zum Risikomanagement. Die Durchführung setzt auf „strukturiertes Brainstorming“ und intensive Modellierung.

Der CORAS-Ansatz hat drei Bestandteile: (1) die (graphische) Sprache, die mehrere Diagrammtypen für die Risikomodellierung zur Verfügung stellt, (2) die übergreifende Methode, um Risiken zu identifizieren und zu bewerten, und (3) ein erstes Modellierwerkzeug (Editor) auf Eclipse-Basis.

Die Methode definiert insgesamt acht Arbeitsschritte, die sich wie folgt in drei grobe Phasen einteilen lassen:

1. *Kontext-Ermittlung* (Schritte 1–4): Gegenseitige Einführung in Methode und Zielsystem durch Analysten und Auftraggeber. Erstanalyse der Schutzziele (assets).
2. *Risikoanalyse* (Schritte 5–7): Risiken identifizieren, abschätzen und bewerten.
3. *Risikobehandlung* (Schritt 8): Gegenmaßnahmen finden und bewerten.

CORAS bietet eine eigene Diagrammsprache, die sich an UML-Use-Case-Diagramme und die Idee von Misuse-Case-Diagrammen anlehnt. Insgesamt fünf Diagramm-Typen stehen zur Verfügung: Asset, Threat, Risk, Treatment und Treatment-Overview. Die Diagrammtypen bauen aufeinander auf, d. h. sie werden in den Arbeitsschritten sukzessive abgeleitet und ergänzt. In den Diagrammen werden kausale Zusammenhänge und Abfolgen modelliert, wie ursprüngliche Bedrohungen auf die zu schützenden Werte einwirken können. Abbildung 1 zeigt die zur Verfügung stehenden Modellierungsmittel im logischen Zusammenhang. Im Einzelnen sind das:

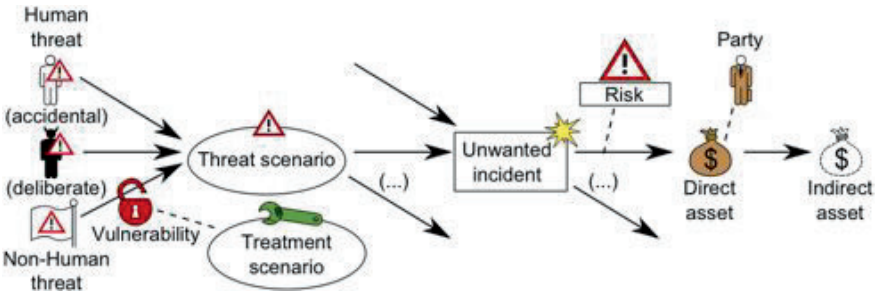


Abbildung 1: CORAS-Symbole im Überblick (©: CORAS [LSS11])

- Alle *Bedrohungen* (threats) gehen von Angreifern aus und sind entweder absichtlicher, unabsichtlicher oder technischer Art.
- Angreifer nutzen *Schwachstellen* (vulnerabilities), um sich Zugriff zu einem System zu verschaffen.

[1] Schutz von vorhandenen Werten, im Gegensatz zur Abwägung von Handlungsoptionen („offensiv“)

- Ein *Bedrohungsszenario* (threat scenario) beschreibt die Art oder den Ablauf eines Angriffs. Es fasst eine Folge von Ereignissen zusammen, die von einer Bedrohung ausgelöst wird und einen Beitrag zu einem unerwünschten Ereignis leistet. Szenarien können weiter zu einem Netzwerk aufgesplittet werden, um das mögliche Zusammenwirken genauer zu erfassen.
- Durch den Angriff wird ein *unerwünschtes Ereignis* (unwanted incident) ausgelöst, z.B. eine Gefährdung tritt ein. Unter Berücksichtigung einer Wahrscheinlichkeit und eines Schadensausmaßes erwächst daraus ein *Risiko* (risk).
- Durch das unerwünschte Ereignis sind unmittelbar die *direkten Schutzziele* (assets) wie Vertraulichkeit, Systemintegrität, usw. bedroht und mittelbar aber auch *indirekte Schutzziele* (z. B. Reputation).
- Eine *Partei* (party) entspricht einem Stakeholder, der ein besonderes Interesse an einem oder mehreren Schutzzielen hat.
- *Gegenmaßnahmen* (treatment szenarios) repräsentieren die Mittel und Abläufe, um Schwachstellen zu behandeln.

Verschiedene High-Level-Modellierungskonzepte und eine formale Semantik komplettieren die CORAS-Methode.

2.2 Behandlung von Wahrscheinlichkeiten in CORAS

CORAS unterstützt die qualitative und quantitative Bewertung der modellierten Bedrohungen. Ziel ist es, die Risiken zu bewerten, die von den unerwünschten Ereignissen ausgehen. Dazu muss für jedes dieser Ereignisse das zu erwartende Schadensausmaß und die Wahrscheinlichkeit oder die Eintrittshäufigkeit bestimmt werden. In Anbetracht der schwierigen Direkt-Abschätzung wird vorgeschlagen, die quantitative Bewertung anhand einer schrittweisen Berechnung auf Basis des Bedrohungsdiagramms vorzunehmen (siehe [LSS11], Kap. 13). Als Best-Practice wird Folgendes vorgeschlagen:

- Den Kanten (initiates-/leads-to-Beziehungen) im Threat-Diagramm soll jeweils eine Wahrscheinlichkeit zugeordnet werden. Der Wert gibt die *bedingte Wahrscheinlichkeit* für eine erfolgreiche Fortsetzung des Angriffs entlang dieser Kante an.
- Für die Knoten, die Bedrohungsszenarien und die unerwünschten Ereignisse, kann dann die Wahrscheinlichkeit oder Häufigkeit abgeleitet werden. Dabei gibt der errechnete Wert an, wie wahrscheinlich - unter den gegebenen Kantenwahrscheinlichkeiten - das Erreichen eines Knotens über einen beliebigen gerichteten Pfad im Threat-Diagramm ist.

Randbedingungen für die Wahrscheinlichkeitsberechnung sind die statistische Unabhängigkeit oder der wechselseitige Ausschluss der Szenarien auf eingehenden Pfaden. Auch die (Un-) Vollständigkeit der Analyse ist zu berücksichtigen. Abbildung 2 zeigt ein Beispiel.

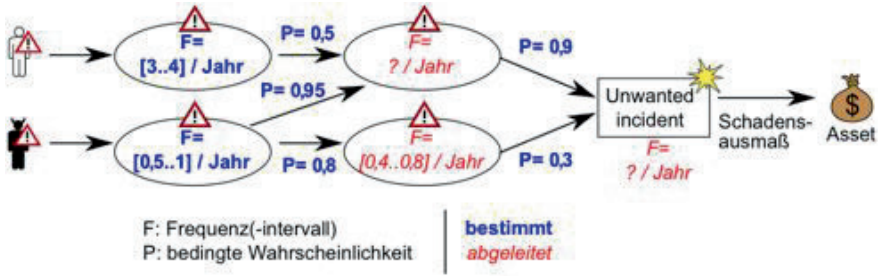


Abbildung 2: Beispiel für die Berechnung von Wahrscheinlichkeiten nach CORAS

3 Einflussfaktoren für die Häufigkeitsbewertung

Die CORAS-Methode ist ein allgemeiner Ansatz zur Risikoanalyse, der in vielen Domänen einsetzbar ist. Allerdings verlangt diese Allgemeinheit auch methodische Verfeinerungen bei der Anpassung an spezielle Aufgabenbereiche.

3.1 Strukturiertes Vorgehen bei der Risikoidentifikation

Im Prozess der Risikoanalyse werden mögliche IT-Security-Bedrohungen für das untersuchte System identifiziert. CORAS verwendet in dieser Phase *Threat-Diagramme*, um den Verlauf und die Auswirkungen auf die festgelegten Schutzziele darzustellen.

Für die Anwendung im Bereich der Eisenbahnsignaltechnik, in der häufig komplexe und verteilte Systeme vorhanden sind, bietet es sich an, den Verlauf eines Signals zu verfolgen: Beginnend an dem Ort des Entstehens (z. B. Sensor am Gleis, siehe Achszähler-Fallstudie in Abschnitt 4) bis zur Verarbeitung der Informationen (z. B. im Achszählrechner). Bei jeder zwischenzeitlich durchlaufenen Komponente muss hinterfragt werden, ob und – wenn ja – wie eines der festgelegten Schutzziele verletzt werden kann. Diese Pfade werden im *Threat-Diagramm* festgehalten, das als Basis für eine Bewertung dienen kann.

3.2 Ein neuer Klassifikationsansatz zur Häufigkeitsbewertung

Das Ziel der Risikoanalyse ist eine Klassifikation für vorhandene Risiken. Das Risiko ist wie auch in risikobasierten Ansätzen zur funktionalen Sicherheit [Int06] definiert als

$$\text{Risiko} = \text{Schadensausmaß} \times \text{Häufigkeit}.$$

Die CORAS-Methode fordert entsprechend, dass jedes unerwünschte Ereignis (*Unwanted Incident*) in einem *Threat-Diagramm* mit einer Wahrscheinlichkeit oder Häufigkeit (Likelihood) und einem Schadensausmaß zu versehen ist. Eine Klassifizierung der zu erwartenden Schäden bei bestimmten Ereignissen ist relativ leicht möglich und ist im Bereich der Safety-Analysen bereits etabliert.

Es gibt auch in der Eisenbahnsignaltechnik bereits Ideen, die Risikoberechnung statt auf Wahrscheinlichkeiten auf mehreren beitragenden Faktoren aufzubauen [BS12]. Diese Idee wird hier übernommen und ein neues Bewertungsmodell entworfen, das sogenannte *Häufigkeitslevel* definiert. Die Bestimmung erfolgt durch Abschätzung von drei Parametern:

- **Parameter Motivation (m)**: Beschreibt die *Motivation* eines potentiellen Angreifers, eine gewisse Handlung auszuführen, um ein bestimmtes *Angriffsziel* zu erreichen. Damit ist auch ein persönlicher *Gewinn* verbunden, den sich der Angreifer von seinem Handeln verspricht.
- **Parameter Werkzeuge (w)**: Kategorisiert die für die Handlungen des Angreifers erforderlichen *Geräte*, *Mittel* oder auch *Fähigkeiten*, die für die Umsetzung seines Vorhabens erforderlich sind. Das beinhaltet auch finanzielle Mittel und Know-how.
- **Parameter Zugang/Gelegenheit (z)**: Bewertet die Zugänglichkeit des Systems für einen Angreifer, der ein bestimmtes Threat Scenario ausführen will. Dazu zählen eventuelle Zugangskontrollen, Hindernisse, Zeitbedarf, Entdeckungsgefahr und andere Risiken für den Angreifer.

Die qualitative Bewertung kennt für jeden dieser Parameter sechs Stufen (von 0 bis 5), wobei höhere Zahlen für wahrscheinlichere Angriffe stehen. Die Null ist ein Sonderfall und bedeutet „nicht vorhanden“ oder „unmöglich“. Die Motivation wird dabei ausgenommen, da sie niemals gänzlich ausgeschlossen werden kann. Die Tabelle 1 gibt einen Überblick über die Stufen. Die Kalibirierung der Kategorien für die einzelnen Parameter („Wann spricht man von einer *hohen* Motivation oder einer *unwahrscheinlichen* Möglichkeit?“) ist pragmatisch für jede einzelne Anwendung vorzunehmen. Für andere Anwendungsbereiche oder vertiefende Analysen sind durchaus weitere Einflussfaktoren oder alternative Interpretationen denkbar.

Kategorie/ Häufigkeitslevel	**Motivation/Ziele (persönl. Gewinn)**	**Mittel, Werkzeuge, Fähigkeiten**	**Gelegenheit (Zugang, Möglichkeit)**
0	–	unmöglich	unvorstellbar
1	sehr gering	sehr schwierig	unwahrscheinlich
2	gering	schwierig	selten
3	mittel	mittel	mittel
4	hoch	einfach	gelegentlich
5	sehr hoch	sehr einfach	wahrscheinlich

Tabelle 1: Kategorien der einzelnen Parameter zur Likelihood-Bewertung

3.2.1 Berechnung eines Gesamthäufigkeitslevels

Von diesen Parametern ausgehend wird ein Gesamthäufigkeitslevel $h(m, w, z)$ abgeleitet, der für die Risikoberechnung (s. o.) genutzt werden kann. Der Wertebereich von h soll dazu wieder sechs Stufen umfassen (0 bis 5). Außerdem sollen die Parameter in gleicher

Gewichtung eingehen, d. h. dass kein Einfluss dominant hervorsticht. Schließlich soll eine Einzelbewertung von Null („unmöglich“) zu $h(m, w, z) = 0$ führen, was die Sonderstellung dieser Stufe betont.

Für die Gewichtung h wurden verschiedene Varianten untersucht [Saa12], u. a. Mittelwert-, Min/Max- oder Wurzelfunktionen. Am besten geeignet für die Fallstudie scheint der quadratische Mittelwert (s. Eq. (1)) mit Sonderregelung für die Stufe 0, der durch die Betonung von Ausreißern eine leicht „pessimistische“ Einschätzung bzgl. der erwarteten Gesamthäufigkeit aufweist. Für andere Anwendungen ist die Gewichtung h anzupassen. Das Ergebnis wird in einer Risikomatrix (Häufigkeitslevel $\times$ Schadenslevel) genutzt, um zu bestimmen, ob das Risiko noch akzeptabel ist.

$$h(m, w, z) = \begin{cases} 0, & \text{falls } w = 0 \ \vee \ z = 0 \\ \left\lfloor \sqrt{\frac{m^2+w^2+z^2}{3}} \right\rceil, & \text{sonst} \end{cases} \tag{1}$$

3.2.2 Anwendung im Threat-Diagramm

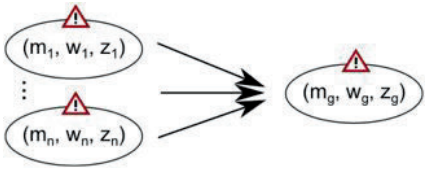
Für die Risikoberechnung mittels CORAS-Threatdiagrammen bedeutet unser neuer Ansatz, dass die Kausalkette eines Angriffs, von der Bedrohung über eine Folge von Bedrohungsszenarien zum unerwünschten Ereignis, nun anders bewertet wird. Für die Knoten im Netzwerk wird statt einer Wahrscheinlichkeit oder einer Eintrittshäufigkeit jeweils ein Häufigkeitslevel bestimmt, was durch Abschätzen der drei Parameter Motivation, benötigte Werkzeuge/Mittel und Zugang/Gelegenheit erreicht wird. Auf (Übergangs-) Wahrscheinlichkeiten an Relationen (vgl. Abschnitt 2.2) wird bei dieser Erweiterung verzichtet.

Die Bewertung läuft iterativ ab. In jedem Schritt muss ein lokales Tupel der Einflussfaktoren (m_g, w_g, z_g) bestimmt werden, und zwar relativ zu den Bewertungen (m_i, w_i, z_i) der unmittelbaren Vorgängerknoten ($i \in [1..n]$). Dazu findet zunächst eine „Initialisierung“ statt: jedem *Threat* wird eine spezifische Motivation (m) zwischen 1 und 5 zugewiesen (sofern zutreffend, sonst „mittel“). Die Anforderungen für benötigte Werkzeuge/Mittel (w) und Zugang/Gelegenheit (z) sind anfangs minimal (höchste Stufe 5). Die eigentliche Bewertung beginnt bei den ersten *Threat Scenarios*, die direkt mit einem *Threat* in Verbindung stehen und nur eingehende *initiates*-Kanten haben.

Dabei lautet die intuitive Bedingung, dass der Angriff durch die Fortsetzung nicht leichter werden kann. Folglich dürfen die Bewertungen des vorhergehenden Knotens nur übernommen oder reduziert werden. Mehrere eingehende Kanten werden als Oder-Verknüpfung interpretiert.[2] Die Bewertung sollte sich am leichtesten Pfad (mit den höchsten Einstufungen) orientieren. Uneindeutige Fälle müssen abgewogen werden, wobei die Maxima der Parameter m/w/z eine theoretische Obergrenze (siehe Abb. 3) vorgeben. Unabhängig davon ist immer eine situationsabhängige Korrektur nach unten möglich, wenn der Angriff in der Fortsetzung als „signifikant schwieriger“ bewertet wird.

Am Ende der Iterationen besitzen alle *Unwanted Incidents* eine Bewertung in (m, w, z)-Form. Aus diesen Parametern wird mit der im Abschnitt 3.2.1 aufgestellten Formel der

[2] Ein Angriff verläuft nur auf einem der Pfade. Parallele/koordinierte Angriffe werden nicht betrachtet.



$$m_g \leq \max_i m_i \,,$$
$$w_g \leq \max_i w_i \,,$$
$$z_g \leq \max_i z_i$$

Abbildung 3: Iterative Bestimmung der Level für die Einflussfaktoren

Gesamthäufigkeitslevel errechnet. Im Anschluss werden – wie in der CORAS-Methode vorgesehen – die Schadensausmaße festgelegt und für die verbundenen Assets die Risiken eingestuft (mittels Risikomatrix).

4 Fallstudie „Achszählsystem“

Im Folgenden wird exemplarisch ein Achszählsystem hinsichtlich seiner Security-Eigenschaften mit der vorgestellten Methode untersucht.

4.1 Überblick

Das Achszählsystem ist eine wichtige Komponente, die im Zusammenspiel mit anderen Einrichtungen die Sicherheit des Eisenbahnverkehrs gewährleistet. Mit diesem System wird das Freisein von Gleisabschnitten – den sogenannten Gleisfreimeldeabschnitten (GFM-A) – überprüft, so dass das Befahren eines besetzten Gleises durch die Stellwerkslogik verhindert werden kann. Das untersuchte Achszählsystem besteht grundsätzlich aus zwei Teilsystemen:

- **Außenanlage**: Am Gleis befinden sich ein Radsensor (RS), ein Gleisanschlussgehäuse (GAG) und die Kabelverbindung zum Stellwerksgebäude. Anschlussgehäuse und Sensor heißen zusammen auch Zählpunkt (ZP).
- **Innenanlage**: Dieser Teil ist im Stellwerksgebäude angeordnet und setzt sich aus einem oder mehreren Achszählrechnern zusammen. Diese verarbeiten die über die Kabelverbindung der Sensoren eingehenden Signale und werten daraus den Belegungszustand eines GFM-A aus.

In einer Bachelor-Arbeit [Saa12] wurde das Gesamtsystem untersucht. Hier soll die Außenanlage in Auszügen betrachtet werden, um die Anwendbarkeit der vorgestellten Anpassungen der CORAS-Methode darzulegen.

4.2 Modellierung der Außenanlage in einem Threat-Diagramm

Die Kontext-Ermittlung (Phase 1 der CORAS-Methode, siehe Abschnitt 2) liefert die zu schützenden Assets, hier die Systemverfügbarkeit (availability) und Systemintegrität (integrity) bzw. Korrektheit einer Funktion. Die Vertraulichkeit (confidentiality) spielt im vorliegenden Fall keine Rolle. Danach werden die möglichen Bedrohungen (Angreifer), Bedrohungsszenarien und unerwünschten Ereignisse identifiziert.

In Vorbereitung auf die auf Einflussfaktoren basierende Bewertung zur Häufigkeitsabschätzung können bei Bedarf noch die Parameterstufen angepasst werden. Die Stufe 2 soll u. a. für spezielle, aber beschaffbare Werkzeuge/Mittel stehen, die Stufe 5 u. a. für längeren ungehinderten Zugang. Die Stufe 0 entspricht höchsten Anforderungen oder Hindernissen, die die Umsetzung eines Szenarios (nahezu) unmöglich machen.

Die Bewertung wird für ein generisches Achszählsystem durchgeführt. Für exponierte Installationen unter extremen Bedingungen (z. B. bei Castortransporten) ist ggf. eine gesonderte IT-Security-Analyse durchzuführen, wobei einer starken politischen, und damit höheren Motivation einerseits, aber auch einem drastisch verstärkten Anlagenschutz andererseits, Rechnung getragen wird.

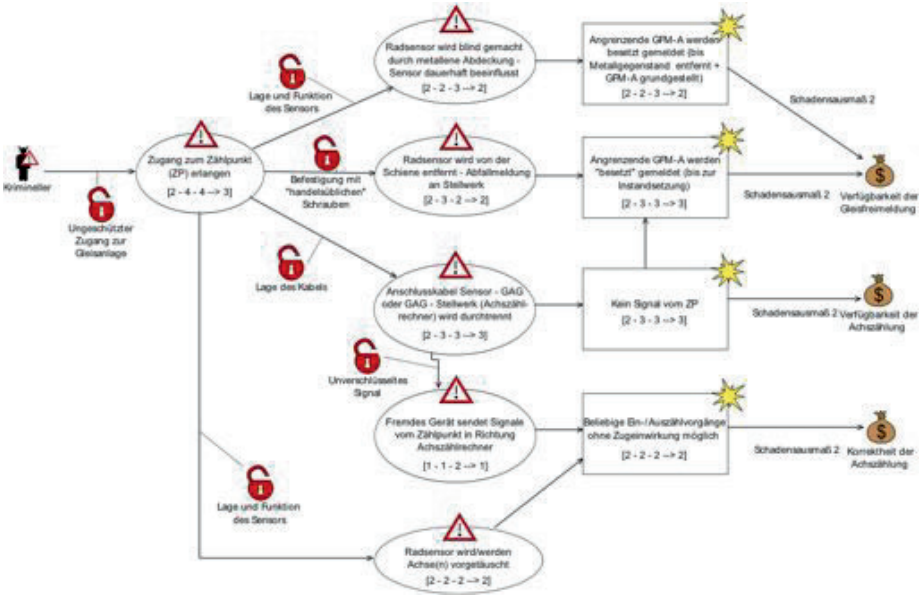


Abbildung 4: Threat-Diagramm der Außenanlage

4.2.1 Identifizierte Bedrohungen

Das Verfahren, dem Signalverlauf in der Anlage zu folgen (s. Abschnitt 3.1), liefert – unter Berücksichtigung der Schutzziele Korrektheit und Verfügbarkeit – ein Threat-Diagramm wie es in Abbildung 4 zu sehen ist. Der Signalverlauf ist dabei so, dass das Signal im Radsensor entsteht – durch eine erkannte Achse – und durch Kabel zum GAG und dann weiter in Richtung Stellwerk (Innenanlage) geleitet wird.

Im Diagramm erkennt man einen „kriminellen“ Angreifer, der beabsichtigt, die Außenanlage zu manipulieren. Zunächst muss die Außenanlage (Bahnstrecke) betreten werden, um Zugriff zum System am Gleis zu erhalten. Von hier aus gibt es mehrere Pfade zu verschiedenen Szenarien.

Der Angreifer könnte eines der Anschlusskabel durchtrennen oder von einem Anschluss abklemmen. Die Lage des Kabels – meist neben den Schienen oder in einem oberirdischen Kabelkanal – begünstigt diese Handlung. Die Nicht-Verfügbarkeit des Zählpunktes (ZP) für das Achszählsystem (*Unwanted Incident* „kein Signal vom ZP“) führt zu einer Besetztmeldung angrenzender Gleisfreimeldeabschnitte, da das System in den sicheren Zustand übergeht. Dies ist eine Beeinträchtigung der „Verfügbarkeit der Achszählung“.

Der Angreifer könnte aber auch weitergehen und, Kenntnisse vorausgesetzt, mit einem selbst hergestellten Gerät die Signale eines Zählpunktes imitieren (Beeinflussungszustand). Letzteres ermöglicht beliebiges Ein- und Auszählen von Achsen für einen Gleisfreimeldeabschnitt, sofern der GFM-A nach der Verbindungsunterbrechung durch den Fahrdienstleiter im Stellwerk grundgestellt wurde. Begünstigt wird dieses Szenario dadurch, dass sowohl zwischen Radsensor und dem Anschlussgehäuse als auch auf der Übertragungsstrecke zum Achszählrechner ein unverschlüsseltes, analoges oder digitalisiertes Signal verwendet wird.

Weitere Angriffspfade, die sich zum Teil auch überschneiden, können aus dem Diagramm abgelesen werden.

4.2.2 Zuweisung der Häufigkeitslevel

Im vorliegenden Diagramm (s. Abb. 4) sind die Parameter zur Bestimmung des Häufigkeitslevels sowie die Schadensausmaß-Kategorien bereits eingetragen. Die Notation der Parameter folgt dem Schema:

[Motivation - Werkzeuge - Gelegenheit --> errechneter Häufigkeitslevel]

Die Bewertung folgt dem Ablauf aus Abschnitt 3.2.2. Dem *Kriminellen* (Threat) wird eine mittlere Motivation (Stufe 3) zugewiesen. Werkzeuge und Zugang werden jeweils mit Stufe 5 initialisiert. Links beginnend wird zuerst das Szenario „Zugang zum Zählpunkt (ZP) erlangen“ bewertet:

Das Betreten der Außenanlage (Bahnstrecke) ist vergleichsweise leicht möglich, da sie nur einfache Barrieren besitzt (Absperrungen, Warn- und Verbotshinweise). Für den Zugang ist eine gewisse „Agilität“ (w=4) notwendig. Das System am Gleis hat keinen zusätzlichen

Schutz gegen unberechtigten Zugriff. Die zu erwartende Zugfrequenz schränkt jedoch die Gelegenheit (z=4) ein. Verbot und Gefahr erzeugen nur eine leichte Abschreckung (m=2) relativ zur Ausgangsmotivation. Insgesamt wird das Szenario mit (2-4-4) bewertet.

Mit dieser Bewertung wird nun die Betrachtung nachfolgender *Threat Scenarios* möglich, wovon wir „Anschlusskabel RS – GAG oder (...) wird durchtrennt“ auswählen. Es bedarf keiner großen Kenntnis der Eisenbahnsicherungstechnik, um zu erkennen, dass eine mit Kabeln angeschlossene Einrichtung wichtig für den Betriebsablauf sein wird. Allerdings wird zum Trennen des Kabels ein Hilfsmittel (w=3) benötigt. Der Zeitaufwand und die Arbeit direkt am Gleis erschweren den Zugang (z=3). Die Motivation ändert sich nicht (m=2). Das führt zu einer Bewertung des Szenarios mit (2-3-3).

Anders verhält es sich beim nächsten Szenario „Fremdes Gerät sendet Signale (...)“. Für diese Aktion ist die Fertigung eines speziellen Geräts notwendig, das die gesendeten Daten eines Zählpunktes imitiert. Dieser Angriff erfordert erheblich mehr Vorbereitung und technische Kompetenzen als die vorausgehenden Szenarien, u.a. Expertenwissen über die Anlage und den Betrieb (w=1). Die Gelegenheit wird herabgestuft, da die Installation eines eigenen Gerätes einige Zeit in Anspruch nehmen wird (z=2). Währenddessen besteht die Gefahr, von einem vorbeifahrenden Zug gesehen oder gar erfasst zu werden. Auch ein durch den Fahrdienstleiter herbeigerufenes Instandsetzungsteam die Manipulation entdecken (nach dem Verlust des Signals vom ZP und dem Übergang in den sicheren Zustand und nach erfolglosen Grundstell-Versuchen des Gleisfreimeldeabschnitts).

Die Motivation für das Threat-Scenario wurde auf 1 reduziert. Es ergibt sich zwar die Möglichkeit für einen Angriff auf die Korrektheit der Achszählung – mit einem „manipulierbaren Zählpunktimitator“ kann nach dem Einfahren eines Zuges in einen Gleisabschnitt dieser *scheinbar* am Ende ausgezählt werden – die Achszählung gibt dann nicht mehr den realen Belegungszustand wieder. Allerdings wird die Manipulation an einem *einzelnen Zählpunkt*[3] durch verschiedene Konsistenzprüfungen (Signalverläufe, Abgleich mit benachbarten Zählpunkten) sicher aufgedeckt. Insgesamt wird das Szenario daher mit (1-1-2) bewertet.

Analog wird bei der Bewertung der Häufigkeitslevel der anderen *Threat Scenarios* verfahren. Es verbleiben noch die *Unwanted Incidents*: Hier wurden die Bewertungen unverändert übernommen, da in dieser Fallstudie die unerwünschten Ereignisse stets eine direkte Folge der vorangehenden Szenarien sind. In zwei Fällen wurden die leichteren Pfade (mit den höheren Bewertungen) ausgewählt.

4.2.3 Schadensausmaße

Für das Achszählsystem werden folgende Kategorien für das Schadensausmaß festgelegt: 0 entspricht einer korrekten Meldung des Ist-Zustands, 1 beschreibt kurze Inkorrektheiten, die zum sicheren Zustand führen, und Kategorie 2 eine länger anhaltende, inkorrekte Achszählung, die auch in den sicheren Zustand führt und ggf. kleinere Reparaturen erfordert. Die höheren Kategorien 3 und 4 sind einer inkorrekten Achszählung vorbehalten, die in einen unsicheren Zustand führen (können).

[3] Für die Security-Analyse der übergeordneten Gleisfreimeldung ist auch die Manipulation von hintereinanderliegenden Zählpunkten zu betrachten.

Für alle *Unwanted Incidents*, die die Verfügbarkeit betreffen, wurde die Kategorie 2 gewählt. Diese Ereignisse verursachen Schäden, die innerhalb der Reparaturzeit behoben werden können, da sie vergleichbar mit einem möglichen technischen Defekt und dem damit verbundenen Austausch von Bauteilen sind.

Das Schadensausmaß für das Ereignis „Beliebige Ein-/Auszählvorgänge ohne Zugwirkung möglich“, das die Korrektheit der Achszählung beeinträchtigt, ist auch in der Kategorie 2 angesiedelt, da – wie zuvor beschrieben – das Auszählen von Achsen und die Freimeldung eines besetzten Abschnittes auf Ebene des einzelnen Achszählers nicht zu einer weitergehenden Gefährdung führt und die inkorrekte Achszählung aufgedeckt wird.

4.3 Analyseergebnisse

Die Beurteilung der Risiken erfolgt aus den Kategorien für das zu erwartende Schadensausmaß und den Häufigkeitslevel, wie in Tabelle 2 dargestellt. Diese Risikomatrix ist als *provisorischer Entwurf* zu verstehen, da es diesbezüglich in der Eisenbahnsignaltechnik noch keine einheitlichen Vorgaben für IT-Security gibt und die fundierte Ableitung einer solchen Matrix weit über den Rahmen dieses Beitrags hinausgeht. Eine Validierung außerhalb der Fallstudie steht noch aus.

In der üblichen Interpretation gilt das Risiko für die grünen Felder als akzeptabel, für die roten als inakzeptabel und für die gelben ist es eine Ermessenentscheidung im Einzelfall. Da die Schadensausmaße 1 und 2 nicht in einen unsicheren Zustand führen, wurde entschieden, dass das Risiko auch für höhere Häufigkeitslevel noch im gelben Bereich ist oder akzeptiert werden kann. Die Bewertungen aller vier Unwanted Incidents liegen hier im grünen Bereich (Schadensausmaß 2 und Häufigkeit 2 bzw. 3, siehe Abbildung 4)

		Schadensausmaß				
		0	**1**	**2**	**3**	**4**
Häufigkeitslevel	**0**					
	1					
	2					
	3					
	4					
	5					

Tabelle 2: Entwurf einer Risikomatrix zur Fallstudie Achszählsystem

Das interessanteste Ergebnis ist die Bewertung des unerwünschten Ereignisses „Beliebige Ein-/Auszählvorgänge ohne Zugwirkung möglich“: Der Wert 2 für die Häufigkeit *und* das Schadensausmaß spiegelt wider, dass trotz des hohen Aufwands für einen Zählpunktimitator kein größerer Schaden erreicht werden kann. Dieser Teil der Analyse rechtfertigt, dass auf weitere Schutzmaßnahmen wie die Verschlüsselung von Signalen verzichtet werden kann.

5 Diskussion

Zur Berechnung des Risikos für die Schutzziele müssen, wie in Abschnitt 3.2 beschrieben, neben dem zu erwartenden Schadensausmaß auch die Auftrittshäufigkeiten der möglichen Bedrohungen bestimmt werden. Mit quantitativen Wahrscheinlichkeiten bzw. Frequenzen sind seltene Ereignisse (nicht nur in der Eisenbahnsignaltechnik) schwer abzuschätzen. Leichte Variationen der Randbedingungen oder alternative Interpretationen können schnell zu Abweichungen von mehreren Zehnerpotenzen führen, welche sich durch die Multiplikation von Einzelwerten noch verstärken. Obwohl die Plausibilität des Gesamtergebnisses kaum überprüft werden kann, entsteht ein trügerisches Gefühl „exakter“ Berechnungen.

In Verbindung mit CORAS haben wir uns mit den auf Einflussfaktoren basierenden Häufigkeitsleveln für einen qualitativen Ansatz entschieden, der sich mit seiner groben Stufeneinteilung an den Grad der Genauigkeit und Verfügbarkeit von Referenzdaten anpasst. Dabei haben wir uns auf drei Hauptfaktoren (Motivation, Werkzeuge/Mittel, Zugang/Gelegenheit) konzentriert, wie sie z.B. auch von der *OCTAVE*-Methode für IT-Security-Risikomanagement verwendet werden (siehe [AD02] Kap. 9.5) Andere Ansätze, z.B. für die Netzwerksicherheit, verwenden spezialisierte Faktoren wie die Reproduzierbarkeit oder Detektierbarkeit von Angriffen auf bekannte Schwachstellen. Unsere Methode kann bei Bedarf um neue Einflussfaktoren und eine neue Gewichtungsfunktion erweitert werden.

Nach einer Normalisierung können positiv-antreibende Faktoren (Motivation $(+)$) und negativ-hindernde Faktoren (benötigte Werkzeuge $(-)$, Zugang $(-)$) zu einem Häufigkeitslevel verrechnet werden. Das heißt, dass während des Bewertungsprozesses die verwendeten Parameter jeweils von hohen Stufen (sehr große Vorteile, sehr geringe Nachteile) iterativ nach unten (sehr geringe Vorteile, sehr große Nachteile) angepasst werden. Eine umfangreiche Abwägung wird auch in [BS12] verwendet, wo der Nutzen (aus Sicht des Angreifers) aus den Erwartungen bzgl. Gewinn $(+)$, Aufwand $(-)$, eigenem Risiko $(-)$ und weiteren Faktoren bestimmt wird. Im Rahmen der Fallstudie hat sich gezeigt, dass der auf Einflussfaktoren basierende Ansatz die Bewertung deutlich transparenter macht, und die Auswahl und Priorisierung von Schutzmaßnahmen erleichtert.

In der praktischen Anwendung überwältigt CORAS zunächst mit seiner Vielzahl an Diagrammtypen (Asset-, Risk-, Treatment-Diagramme usw.), wovon hier nur das wichtige Threat-Diagramm verwendet wurde. Bei den anderen Diagrammtypen handelt es sich im Wesentlichen um alternative *Sichten* mit gemeinsamer Symbolik. Mit der graphischen Modellierung lassen sich Abhängigkeiten zwischen Bedrohungen, Szenarien und Assets gut visualisieren. Ab einer gewissen Systemgröße wird jedoch eine Aufteilung auf mehrere Diagramme notwendig, z. B. getrennt nach Schutzzielen. CORAS besitzt bereits eine Semantik zur hierarchischen Dekomposition von Szenarien. Eine echte Modularisierung der Analyse steht aber noch aus.

6 Zusammenfassung und Fazit

In diesem Beitrag haben wir die Anwendbarkeit der graphischen CORAS-Methode zur IT-Security-Risikoanalyse auf Systeme der Eisenbahnsignaltechnik demonstriert und durch ein alternatives Verfahren zur Abschätzung von Häufigkeiten ergänzt. In einem iterativen Prozess werden Häufigkeitslevel auf Basis von qualitativ abgeschätzten Einflussfaktoren bestimmt, wodurch die Nachvollziehbarkeit der Gesamtergebnisse verbessert wird. Für den Praxiseinsatz ist eine spezifische Kalibrierung der Parameter und ihrer Bewertung sowie eine weitergehende Integration mit existierenden Methoden und Standards notwendig.

Literatur

[AD02] Christopher Alberts und Audrey Dorofee. *Managing Information Security Risks: The OCTAVE Approach*. Addison-Wesley, 2002.

[BS12] Jens Braband und Markus Seemann. On the relationship of hazards and threats in railway signaling. In *Proceedings of The 7th IET System Safety Conference incorporating the Cyber Security Conference, Edinburgh, UK, 15.-18. October 2012*, 2012. (to be published).

[CEN03] CENELEC. EN 50129: Railway applications, Communication, signaling and processing systems – Safety-related electronic systems for signaling, 2003.

[CEN10] CENELEC. EN 50129: Railway applications, Communication, signaling and processing systems – Safety-related communication in transmission systems, 2010.

[Int06] International Electrotechnical Commission. DIN IEC 61508-4: Funktionale Sicherheit elektrischer/elektronischer/programmierbar elektronischer sicherheitsbezogener Systeme, 2006.

[Int09] International Organization for Standardization. ISO 31000: Risk management – Principles and guidelines, 2009.

[LSS11] Mass Soldal Lund, Bjørnar Solhaug und Ketil Stølen. *Model-Driven Risk Analysis: The CORAS Approach*. Springer, 2011.

[Saa12] Sebastian Saal. Erweiterung der CORAS-Methode zur strukturierten Ermittlung von IT-Security-Anforderungen an Systeme der Eisenbahnsignaltechnik. (Bachelorarbeit) Technische Universität Braunschweig, Institut für Theoretische Informatik, 2012.

[VDE13] VDE. VDE 0831-102 (draft): Electric signalling systems for railways – Part 102: Protection profile for technical functions in railway signalling, to be issued, 2013.

Building Secure Systems Using a Security Engineering Process and Security Building Blocks *

Andre Rein, Carsten Rudolph, Jose Fran. Ruiz

(andre.rein, carsten.rudolph, jose.ruiz.rodriguez)@sit.fraunhofer.de

Abstract:
In today's software development process, security related design decisions are rarely made early in the overall process. Even if security is considered early, this means that in most cases a more-or-less encompassing security requirements analysis is made. Based on this analysis best-practices, ad-hoc design decisions or individual expertise is used to integrate security during the development process or after weaknesses are found after the deployment. This paper explains the SecFutur security engineering process with a focus on Security Building Block Models which are used to build security related components, namely Security Building Blocks. These Security Building Blocks represent concrete security solutions and can be accessed via SecFutur patterns on the level of domain-specific models for particular application domains. The goal of this approach is to provide already defined and tested security related software components, which can be used early in the overall development process, to support security-design-decision already while modeling the software-system. Security Building Blocks are discussed in the context of the SecFutur Security Engineering Process with its requirement analysis and definition of security properties.

1 Introduction

Considering security in all phases of a system development process means to introduce an additional view in all phases including among others requirements specification, design decisions, implementation or documentation. Further, in many cases expert knowledge on security topics is essential for the development of secure systems. The SecFutur project [SF] develops a process that provides this additional view through UML-based security models of a system and combines these with support for design decisions based on security building blocks that are also modeled using UML.

Large parts of the SecFutur process use domain-specific artifacts. The domain-independent parts are a core security meta-model defining the concepts used in modeling security and the rather technical security building blocks describing how to use concrete implementations of security functions. Domain-specific parts include domain-specific security models, specific system models and security patterns that describe how security building blocks can be used and combined within a particular application domain to satisfy specific secu-

*This work was supported by the EU FP7 project SecFutur

rity requirements.

The Goal of SecFutur is to provide a modular modeling framework that allows the creation of precise and pluggable representations of the specialized knowledge of different application domains. The design of the representation mechanisms must also take into account the different roles involved. The specialized knowledge must serve different purposes. Of course, it should increase the security of the system. Further, documentation of security requirements and security design decisions is very important in the engineering process and should be tool-supported. Finally, the framework should provide useful knowledge to system engineers (security requirements, threats to consider, available solutions, trust models,...) and help developers to correctly implement and / or integrate security solutions, do optimization, testing, assurance, etc.

This paper describes the SecFutur security engineering process and focuses on the model for security building blocks (SBBs). These SBBs are a core element of the SecFutur process, as they represent domain-independent security functions and help developers to understand how to securely use particular security functions. Furthermore, in addition to interfaces, conditions and information on the functionality, SBBs also provide information on the set of assumptions that needs to be satisfies in the system. These assumptions can then either be realized by additional building blocks or can be subject to risk analysis and concrete threat analyses.

2 State of the Art

A large variety of security technology and individual security solutions exists that can be used by system developers to support security in their systems. Thus, in principle, a developer should be able to take security design decisions early in the development process and just choose from the large set of available security solutions this idealized view of development processes is only valid in very rare cases. The usual approach is a mixture of more-or-less systematic security requirements analysis, ad-hoc design decisions, some best practices, individual security expertise and finally step-by-step improvement after weaknesses have been found either by security testing or in the deployed product.

Another interesting aspect is that only a rather small set of available security solutions is actually used in real-life products and is obviously not available in the current design processes. One prominent example is the Trusted Platform Module (TPM) as specified by the Trusted Computing Group (TCG) [Gro]. A very interesting work [Pea02] describes TPMs as the future elements for security. This security chip is already available in thousands of notebooks and laptops. A TPM potentially can provide various security functionalities that can be used to secure network connections, monitor the security of devices, securely store data, etc. However, only a very small subset of installed TPMs is actually activated and used. Furthermore, if it is used only a very small subset of available functions of the TPM is involved. Experience with developers interested in applying the TPM has shown that one of the reasons for the missing uptake of this technology is the complexity of the specifications. It is not clear how combinations of TPM commands (or TCG software

stack commands) can actually implement some particular security services. Thus, making advanced security functionality available for development processes is a challenge.

A different approach for security solutions are security patterns. Usually, technical details and implementation details necessary for development are not included in the concept of security patterns [Ste06, Roe01]. Some interesting works are the one presented by H. Lohr et al. [LSW10] where he presents patterns for secure boot and secure storage in computer systems, the security patterns for mobile ad hoc networks developed by Jayraj Signh et al. [SSS11], security patterns for agent systems defined by Paolo Giorgini et al. [MGS03] or the work for architecting software with security patterns done by Riccardo Scandariato et al. [SYHJ08]. Although there exist more works for using security patterns as security solution of systems one problem they have is that they do not cover all the different phases of the creation of a system, as they are used for design or implementation but not in the modeling phase. This can create several problems, being one of them the outdating of the models of the system. For example, if, after modeling the system, an engineer uses a security pattern that needs additional elements such as a database or key-store the system will be modified with elements not defined in the modeling phase. These new elements can create new configurations or security flaws that were not expected. For that reason, our approach uses the idea of security patterns and extends it with Security Building Blocks (SBBs). They are represented by UML models in order to describe the functionality and characteristics of security properties in a real world scenario. These SBB models reflect security related software components, which are encapsulated abstractions of program functionalities. Software abstraction, encapsulation and information hiding build the basis of those SBBs. The main focus of using Building Blocks has always been reusability, maintainability and documentation. An interesting work [LSW87] describes these basic concepts with relation to General Building Blocks used in software development. Consequently, this work tries to refine those general concepts and apply them in the field of security, to model and build more secure systems.

The security patterns are not used directly with the security requirements of the system. They are connected to them by means of the security properties defined in the Domain Security Meta-models (explained in the following section). Each security property provides the solution as a security pattern and this one provides a set of SBBs for the implementation of the solution. Thus, the security patterns are not related to the security requirements of the system but to its security properties. This way the modeling phase and implementation phase of a system are related in a direct way and it is naturally embedded in the system.

3 Security Engineering Process

The development of systems composed of embedded components is a very complex task due to their specific characteristics and nature. Many systems of embedded components are composed of a lot of different embedded devices, such as the smart metering system. In these systems, many smart metering devices obtain the information of metering from many houses, process and send it to a different node. This node checks, processes, stores, etc. the

information and sends it to another node, which works with the information provided from many nodes as that one. The systems of embedded components has a reactive nature too. When they process information they may need to react in a specific way, e.g. activating other systems, sending information, etc. One example is the forest control system, where, if they detect a fire, they have to send an alarm. Following this last example we can see that these systems have a real-time nature. They obtain the information, process it and work in real-time. For example, in the Mobile ad-hoc Network (MANET), the nodes enter and exit the system without warning, so the system must react in real-time to these changes and act accordingly. These systems use many components and resources, being hardware or software. For example, they can work with external components such as transmitters, video cameras, sensors or resources such as key-stores, databases, APIs, TPMs, etc.

Through some research done by the companies involved in the SecFutur project and others related to the development of TPM systems, real-time systems, etc. it was learnt that companies usually do not follow a clearly defined engineering process for the development of these systems. Their way of work is start developing and implementing as soon as they can. They sometimes follow a methodology but their focus is to start developing and adding functionalities as they find it necessary. This implies that security is either implemented later in the process as an extra feature or just ignored. Sometimes, when the functionality of the system is almost complete they start adding security. Obviously, in this stage, security is not naturally integrated in the system. Of course, this observations does not hold for systems with strong safety regulations, where strict waterfall development models are in place. In such a clearly structured development process, current threat-based approaches are also not suitable, because the detection of threats and high risks later in the development process requires to start-over and go back to the requirements phase.

SecFutur proposes a security engineering process that allows to develop and use security solutions in order to satisfy the security requirements of systems of embedded components. It integrates, in a flexible way, security solutions in a framework for the development of systems composed of embedded components. Its main objective is to help developers and engineers in the management of security aspects and its use in System Models. The process can be applied to existent processes, improving the security functionality of any process used to model a scenario. Due to size limitations it is not possible to explain all the details and characteristics of the SecFutur Security Engineering Process. A more complete description of the process itself can be found at [RHM11].

Some of the most important characteristics of the process are:

- It helps system developers in making design decisions for finding the best solution for their systems
- It facilitates the certification and the national / international regulations of the security artifacts
- It satisfies the specific requirements of the systems
- The implementation solutions are provided by means of SecFutur Patterns (SFPs) and Security Building Blocks (SBBs), where domain-specific SFPs define how domain -independent SBBs are used and composed within a particular domain.

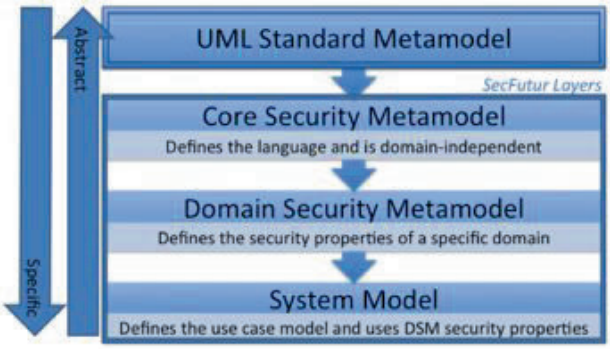


Figure 1: SecFutur Layers

3.1 Artifacts

The different artifacts of the security engineering process have specific objectives and functionalities. Figure 1 describes the artifact structure.

The SecFutur layers go from more to less abstraction and specification. The upper one, the Core Security Meta-model (CSM), is the most abstract. It is based on the UML Standard Meta-model. This meta-model is used as basis for the definition of the different UML elements used in the creation of the CSM such as classes, relations, attributes, etc. The CSM defines the grammar and language for the definition of the domain-specific security artifacts. Because of that, the CSM is domain independent and only defines the abstract architecture. The specification of the security properties and characteristics of each domain is done in the Domain Security Meta-model (DSM). This one uses the CSM as basis because it defines the language. Finally, the System Model is the most specific layer. It is the model of the use case. In this model the system engineer imports a DSM (or various DSMs) and apply its security properties in order to fulfill the security requirements of the system. As we said before, due to the size limitations the reader can find more information of these layers in [RHM11].

The DSMs, as we explained before, define the specification of the domain security knowledge. It allows experts to capture their security knowledge (properties, solutions, threats, etc.) related to specific scopes (standards, company policies, etc.) in a specific domain (Mobile ad hoc Network, Smart meters, etc.). The security properties defined in a DSM are related to implementations by means of SecFutur Patterns (SFP) and Security Building Blocks (SBBs). The security properties define the characteristics and attributes of the solution and the SFP and SBBs their implementation using software / hardware elements. A SecFutur Pattern is a evolved version of the traditional security patterns [Ste06, Roe01], adapted and extended to the SecFutur Engineering Process. It provides information such as the security properties provided and the elements of the system where they can be applied, some examples of use (with support for computer-processing), the elements of the system model that must inter-operate with the pattern elements, a list of restrictions and

metrics of the pattern with regards to the security properties defined before, the elements that must be added to the system in order for the pattern to work (this part is done by using the Security Building Block Models (SBBMs), a series of rules (in OCL format) used to verify the sound integration of the pattern in the system, a series of assumptions that apply to the system once the pattern has been integrated, some known uses and finally related patterns. Due to size limitations we only do a superficial description of this element. Thus, each security property is attached to a SFP, which defines its implementation by means of a SBBM and SBBs. The Security Building Block Models define the structure, relations and elements of the solution. Its basic elements are the SBBs. A SBB (or the aggregations of several SBBs) can provide the implementation solution for a specific property in a specific DSM. Resuming, each security property of a DSM has a SFP that describes its solution, which is implemented by means of SBBs. Following we describe these elements, its characteristics and functionality.

The creation of a DSM involves two different steps. First the analysis of the domain and second the definition of the different security properties. Although the two steps are out of scope of this paper we describe them briefly so the reader can understand better how the Security Building Block Models and the Security Building Blocks provide solutions to a great number of security properties of different domains. Briefly, the analysis of the domain checks the possible security threats and security properties of the system. This analysis provides the necessary information for the definition of the security properties. Once the security domain expert has the information of the domain, she starts modeling security properties. Each security property is composed of several elements such as its threats, assumptions, certifications, V&V (Validation & Verification) elements, etc. After the security domain expert defines a security property she searches for the SBB model that can provide a solution. The search of the solution is done by checking the characteristics of the security property in the list of SBB models. The SBB models are defined by the security property they fulfill, some requirements of the system (such as the elements they need in order to work correctly), the domain, etc. When the SBB model is found, it is attached to the SFP and then linked to the security property. If a combination of SBBs is needed, a more complex SFP needs to be build and validated / verified within the context of the DSM.

4 Security Building Blocks

In contrast to a security pattern, one single Security Building Block does not describe a complex integrated security solution. SBBs should be seen as encapsulated components that are domain-independent and can interact with other components in order to provide a clearly defined security service. The concept of abstracting software functionalities in SBBs can use other SBBs and can also interact with other components in a clearly defined way. In principle, a SBB can be just a concrete implementation of a security solution. However, in order to integrate a SBB into the engineering process, a description of the SBB is required. Here, this description is done in terms of a UML model. Thus, a so-called SBB Model represents one (or several) instantiations (i.e. implementations) of the

SBB.

As SBBs may reflect concrete implementations of a Security Solution, they also need to provide an interface which defines method names and data types used by the SBB. On one hand this is documentation for the system modeler, to better understand how the SBB can be integrated in the system model. On the other hand it serves as a concrete specification how the SBB may be implemented in a concrete realization. In addition to the security properties (or security service) provided by the SBB, this model also needs to provide information on preconditions and constraints, as well as on postconditions on the system that need to be fulfilled after the SBB was applied.

The SBB Meta-model, which is described in detail in Section 4, defines all the different artifacts and their relationships which were concisely presented in this section.

4.1 The Security Building Block Meta-model

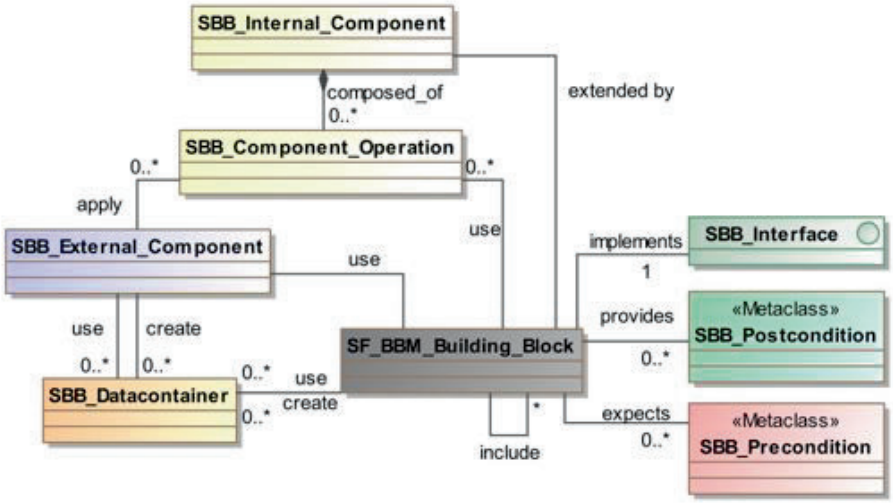


Figure 2: SBB Meta-model

The Security Building Block Meta-model (SBB Meta-model), as shown in Figure 2, delineates artifacts and relationships to construct Security Building Block Models (SBBM). More concretely, the SBB Meta-model acts as a determining factor to depict one or more Security Building Blocks and their interactions with other artifacts to build a SBBM.

Different artifacts enable the SBBM designer to describe their SBBMs in a concrete way (as a detailed view of internal SBBM components), but also leave the possibility to describe interfaces which may be used from a system model. Although a concrete SBBM

and its SBBs may be used in concrete implementations, its main purpose is helping to represent Security Properties at the implementation level. These implementations provide the solutions of the security properties defined in the Domain Security Meta-model. The solutions are specified by means of Security Patterns.

Each SBBM and so their SBBs comes with its own dependencies and conditions. These dependencies may be whole system components (databases, key-chains, TPMs or other external components), simple data structures (cryptographic keys, plain-text / encrypted data, etc.) or even other SBBs. In a later stage of the development of the SecFutur tools, all these dependencies should be resolved automatically and imported into the concrete system model after a Security Pattern is selected as a solution for a specific Security Property.

The following presents a more detailed explanation of the artifacts shown in Figure 2.

4.2 Artifacts and Interactions

4.2.1 SBB_Datacontainer

The SBB_Datacontainer artifact is the most general type in the SBB Meta-model. It is used as a container for any kind of data which needs to be processed by a SBB. It may appear in the model as an output value of an external component (*SBB_External_Component creates SBB_Datacontainer*) or of a SBB (*SF_BBM_Building_Block creates SBB_Datacontainer*). Additionally a SBB may use it as a input value (*SF_BBM_Building_Block uses SBB_Datacontainer*).

4.2.2 SF_BBM_Building_Block

The SF_BBM_Building_Block artifact (SBB) is the key components of any SBBM. SBBs are used to represent any kind of security-related system components and encapsulate them in a single artifact which is used in the SBBM. A SBB may be defined broadly in an early stage of the SBBM and refined more detailed as soon as more information is needed or provided. Since a SBB may be composed of other SBBs the level of detail may be increased during the SBBM development process when it is needed or required. On the other hand it is also common that SBBs are used to compose a more complex SBB. This is also independent from the level of detail of any single involved SBB and depends only of the desired level of abstraction of the SBBM. Any artifact from the SBB Meta-model has at least one direct relationship to a SBB.

4.2.3 SBB_Precondition

A *SBB_Precondition* is an requirement which specifies under what conditions a SBB may be applied successfully (*SBB expects SBB_Precondition*). A single SBB_Precondition represents exactly one requirement, which may be formal or informal. It is designated that

for any different requirement a single artifact instance is used. For example if an input value of a SBB is a cryptographic key the SBB_Precondition may determine that its size must be at least 128Bit. Additionally another SBB_Precondition may determine that a random number for the key generation may only come from a source considered secure (e.g. ”/dev/random” instead of ”/dev/urandom” in Unix Systems). Preconditions can be either satisfied by other SBBs (to be defined in a SecFutur Pattern) or remain as assumptions for the final system that need then to be evaluated for the environment the system should run in. This evaluation of remaining assumptions will be part of a risk analysis. In security certifications, these assumptions express policies for the operational environment.

4.2.4 SBB_Postcondition

A *SBB_Postcondition* is a statement that is valid if a SBB is applied successfully (*SBB provides SBB_Postcondition*). A successful application implies that any SBB_Precondition was obeyed. For example a SBB which encrypts confidential data under a given key may assert that the output data may be protected against eavesdropping. If a SBB has multiple assertions which become valid after a successful application, each different statement must appear as a single artifact instance. Again a SBB_Postcondition may be formal or informal.

4.2.5 SBB_External_Component

A *SBB_External_Component* is a system component which must be available for a SBB to function properly, but lies out of scope of the current SBB or even the SBBM. A SBB may use the functionality of the external component either by using its functionality directly (*SF_BBM_Building_Block uses SBB_External_Component*) or by using data structures that are produced by it (*SF_BBM_Building_Block uses SBB_Datacontainer created_by SBB_External_Component*) . If a Security Pattern is selected which involves external components the system engineer is informed about what specific external components are needed. Either the system engineer must provide these components from within their own system model or they are created automatically represented by additional interfaces or even concrete implementations. Another possibility is that a SBB_Datacontainer is used by a SBB_External_Component as an input value *(SBB_External_Component uses SBB_Datacontainer*).

Additionally an SBB_External_Component may apply functionalities of an SBB_Internal_Component, by using its SBB_Component_Operations (*SBB_External_Component applies SBB_Component_Operation*). For example using a SBB implementation which involves a TPM, many functionalities are based around the reporting of a system state. To keep track of the system state, an external component, namely IMA (Integrity Measurement Architecture) is used. IMA applies an operation of the TPM, which modifies internal registers in the TPM. These registers, which reflect the current system state, are later used in SBBs which need this system state for their own functionality. In consequence it is mandatory to distinguish between external and internal components. Both components must be available, but an external component represents a part of the system where the SBBM or a concrete SBB has no direct influence.

4.2.6 SBB_Internal_Component

The *SBB_Internal_Component* represents a part of the SBBM which is directly associated with a SBB over its SBB_Components_Operations. Any internal component consists of SBB_Component_Operations, which represent a concrete functionality of the component. A SF_BBM_Building_Block may modify a SBB_Internal_Component (*SF_BBM_Building_Block modifies SBB_Internal_Component*), which is considered as an unspecified usage of a SBB_Component_Operation within the SBB.

Additionally a SBB may modify a internal component (*SBB_Internal_Component extended_by SF_BBM_Building_Block*) such that it enhances the functionality of the component. This operation is also not precise and should be explained in detail depending on the enhancement (e.g. with additional diagrams or textual).

4.2.7 SBB_Component_Operation

A *SBB_Component_Operation* is an operation of an internal component which may be executed by a SBB (*SF_BBM_Building_Block uses SBB_Component_Operation*). This usage is handled internally in a SBB and is mostly a call of a function or method, provided by a library of the internal component, which executes the components real operation.

4.2.8 SBB_Interface

The *SBB_Interface* describes the public interface which may be used by a system engineer or other SBBs which need to integrate the current SBB. The application of the SBB is limited to just this specified operations and thus the only way to communicate with the SBB. The SBB_Interface serves as documentation for the input and output values as well as the description of the functionality. This information is mandatory for a system engineer who wants to use SBB in a concrete system model. Additionally the SBB_Interface is used when SBBs are combined with each other. This aspect is described in more detail in Section 5.3 and 5.4.

5 Example Model

Assuming there exists a SBB which simply encrypts data by using a symmetric cipher[1], as shown in Figure 3. This SBB needs at least data which should be encrypted (Data) and, additionally, a key (Key) that is used to apply the encryption. After the data is encrypted by the SBB, an output value is created which contains the original data encrypted under the given key (EncryptedData). A system engineer, who wants to integrate SBB_Encryption in a system model, needs the information about all input and output values of a SBB.

[1]For the used example a concrete encryption algorithm was intentionally left out. An encryption SBB which is used in a real world scenario must always specify a concrete algorithm (like AES-128 in CBC mode) or provide a mechanism to select / propose a proper algorithm.

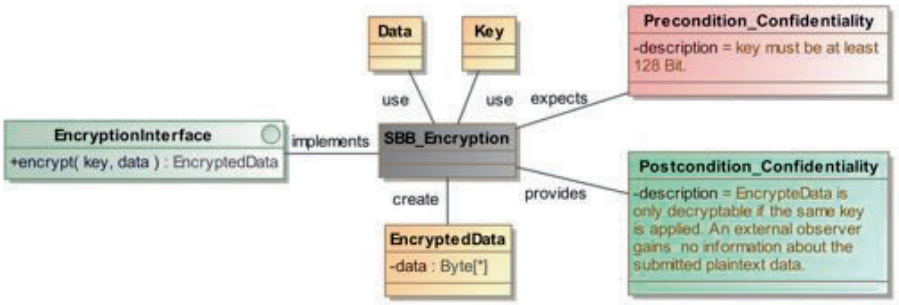


Figure 3: Combining SBBs

Although the SBB Meta-model defines the general type SBB_Datacontainer, it is necessary that the specific supplied input and output data to any SBB is specified concretely in a SBBM.

5.1 Interfaces for SBBs

Each SBB consists of an interface which may be used in a system model to apply the functionality of that specific SBB. In this example the input values `data` and `key` are parameters to the `encrypt()` method. This method results in the output data `EncryptedData`. Figure 3 shows the `EncryptionInterface` of the Encryption SBB. More details on interfaces can be found in Section 5.4.

5.2 Preconditions and Postconditions

Describing a SBB only with its input and output values is insufficient in most cases. Therefore two additional artifacts are used to describe so-called Preconditions and Postconditions. Both conditions are optional and should only be used if there are concrete conditions for the described SBB. Using conditions to describe under which circumstances a software component may be executed and what it provides, is based on the concepts of *Design by Contract*. Two interesting works [Mey92, Mey97] describe how these conditions can be applied in software design and development.

A Precondition always describes restrictions, which need to be fulfilled, before the SBB may be applied successfully. Thus they represent a requirement information for the system engineer. In this example, as shown in Figure 3, the SBB_Precondition states that the key used for the encryption must be at least 128 Bit. If the system uses a key that does not meet this precondition the successful application of the SBB is not guaranteed and therefore the postcondition is not provided. In consequence this means that the system engineer is

forced to fulfill all the preconditions of any used SBB in order to obtain the postconditions they provide.

On the other hand a SBB_Postcondition describes what the SBB provides if applied successfully. In this example the SBB_Postcondition_Confidentiality states that EncryptedData is now encrypted under a specific key and may only be decrypted if the same key is used. Furthermore it states that an external observer is not able to gain any information about the submitted original data.

If a system modeler now uses EncryptedData (e.g. send it to another system component or over a network device), he is assured that no one without the specific key is able to use the submitted data in any way to gain access to its original plain-text content.

5.3 Combining SBBs

It is also possible that SBBs are combined with other SBBs to enhance and encapsulate functionalities. There are different ways to express such a combination. One way is to use domain-specific SFPs to define the combination. If the combination results in another domain-independent security functionality, it can be useful to describe the combination as another (more complex) SBB. In all cases, the combination needs to be be validated or verified by security expert. If detailed enough and if a formal (or operational) semantic is provided, the SBB model can be the foundation for a rigorous verification. The following paragraphs concentrate on the creation of new SBBs by combining existing SBBs.

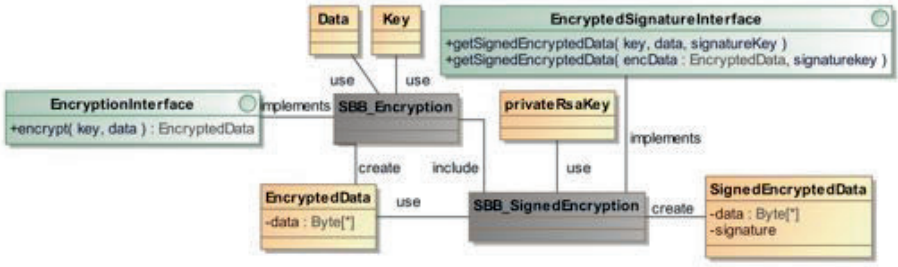


Figure 4: Combining SBBs

Figure 4 shows the case where the output of the Encryption SBB is used in another SBB (SignedEncryption) which generates a signature for that data. The include statements states that the interface for SignedEncryption also accepts the input values from the Encryption SBB. Additionally both SBBs are bound through the `EncryptedData` datastructure. .

Another approach to combine SBBs, as shown in Figure 5, is to use two independent SBBs and include both in an additional SBB. There exist two SBBs, one (Signature) generates a signature of any arbitrary data and the other one (Encryption) encrypts any arbitrary

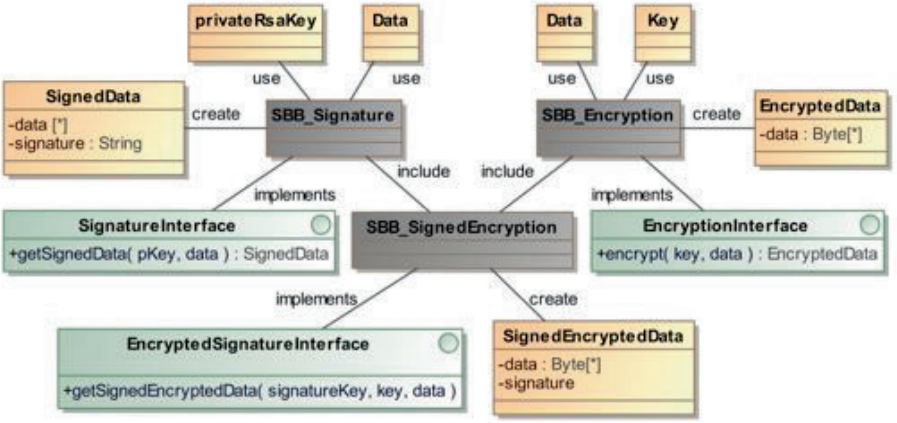


Figure 5: Creating new SBBs by Inclusion

data. Both SBBs may be used independently from another. Both SBBs are included in a third SBB called SignedEncryption. This SBB now uses the SBBs (Signature and Encryption) to generate also an `SignedEncryptedData` data-structure. While in both cases the resulting data-structures are semantically the same, both modeling approaches are different. In the latter case the SignedEncryption SBB is not directly bound to the `EncryptedData` data-structure, which means that `EncryptedData` is no valid input parameter for this SBB by default. (In this example it might be legal to add a separate method which also accepts `EncryptedData` as an input parameter. This decision is left to the SBBM designer and depends on the concrete SBB.)

5.3.1 Aggregation of Conditions

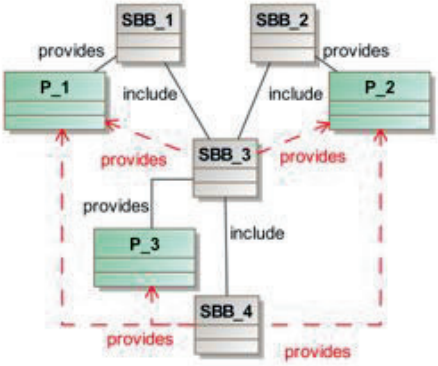
Another important aspect while composing SBBs is that postconditions are aggregated.

Figure 6 shows a general architecture of SBBs and how the aggregation takes place in deeper and nested hierarchical structures.

Whenever a SBB is aggregated of one or more SBBs (SBB_3 includes SBB_1 & SBB_2 and SBB_4 includes SBB_3) the including SBB (aggregate) also provides all postconditions of its included SBBs (parts). The table from Figure 6 shows all SBBs and their provided postconditions, where ”o” marks an indirect aggregated postcondition, ”x” a directly aggregated postcondition and ”-” that no postcondition is adopted.

An aggregate itself may always add additional postconditions, as shown for SBB_3 in Figure 6, but these direct postconditions do not affect the postconditions of its parts.

When working with SBBss in a specific system model, this means that a SBB may be substituted by a SBB which is deeper nested in the same hierarchical aggregated struc-



	P_1	P_2	P_3
SBB_1	x	-	-
SBB_2	-	x	-
SBB_3	o	o	x
SBB_4	o	o	o

Figure 6: Conditional Aggregation

ture (covariant behavior [Car88]). For example SBB_2 may be substituted by SBB_3, but SBB_3 must not be substituted by SBB_1 or SBB_2.

Moreover, it is also possible to substitute a SBB with another independent SBB that is not even in the particular hierarchical aggregation structure (the one of the original SBB). The explanation for this property is that if two distinct and independent SBBs provide the same postcondition, they can be substituted by one another.

For preconditions the same methodology may be used. As long as a particular hierarchical aggregation structure is observed the preconditions of any SBB in that structure stay the same. As soon as a distinct and independent SBB is able to substitute a SBB (by satisfying the original postcondition), the preconditions may be different to any of the preconditions of the substituted SBB. This means that if a SBB provides the same postcondition, but with totally or partially different preconditions, it may be used though.

While composing SBBs it is possible that postconditions are aggregated which contain contradictions. Assuming that the postcondition P_3 is the contrary of P_1 ($P\_3 = !P\_1$). This would mean that SBB_3 has both P_1 and $!P\_1$ as a postcondition.

In general, SBBs provide a process to systematically aggregate postconditions and properties. However, as security is not composable in general it cannot be concluded that the aggregated set of postconditions is actually satisfied for the combined SBB. There is no generic approach to verify these postconditions as this verification strongly depend on the particular properties expressed and on the character of the security functionality represented by the SBB. The SBB expert is responsible to provide evidence that the combination is correct. This evidence can range from best-practice or knowledge of the expert to results of a formal verification.

5.4 Interface from System Model to SBBM

A crucial part in the modeling of SBBs is the definition of an interface. This interface is used either from a system engineer who integrates a SBB into the system model or used to interconnect SBBs within the SBBM itself. The interface is the only visible part and thus the only way to communicate with the SBB. This means that if a SBB is applied in a system model the system modeler may only interact with the specified methods of the building block. While this is the standard procedure how interfaces are used in general, it is a crucial requirement as a SBB expert has to consider the interfaces when designing SBBs.

While a SBB itself provides a solution for a specific security requirement, there may also exist different SBBs solving the same requirement but with different other components involved. As long as the interface of any different SBB is equal, the SBB is easily exchangeable during the development process.

6 Conclusion

The creation of a security model for a system is a very complex task due to the different security requirements, constraints, functionalities, etc. The security engineering process and security building blocks described in this paper helps developers in creating a security model that fulfills all the security requirements of a domain-specific system using security properties and the security building blocks that implement them. Currently, the process is been applied to several use cases of different domains in the SecFutur project, showing good results in each of them.

The SBB Meta-model and the SBB Models as provided in this paper provide one possible approach towards exact specifications of security solutions and their integration into security engineering processes. A validated security solution can be described in a way that preconditions, constraints, dependencies, etc. are exactly expressed and considered in the integration of the SBB into a system.

The next step in this work is to enhance and update the security engineering process with security patterns (that will describe how to create a solution for complex security properties), create certification for the models and increase the DSM online repository with more artifacts. Regarding the SBBs, the next steps are to integrate the concept of SBBs with the SecFutur CSM and DSM and describe domain-specific integrations of SBBs for the realization of more complex security properties.

References

[Car88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[Gro] Trusted Computing Group. TPM Main specification.

[LSW87] M. Lenz, H.A. Schmid, and P.F. Wolf. Software Reuse through Building Blocks. *Software, IEEE*, 4(4):34–42, july 1987.

[LSW10] H. Lohr, A. R. Sadeghi, and M. Winandy. Patterns for Secure Boot and Secure Storage in Computer Systems. 2010.

[Mey92] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, oct. 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2. edition, 1997.

[MGS03] Haralambos Mouratidis, Paolo Giorgini, and Markus Schumacher. Security Patterns for Agent Systems. 2003.

[Pea02] S. Pearson. Trusted Computing Platforms, the next security solution. Technical report, HP Labs, 2002.

[RHM11] Jose Fran. Ruiz, Rajesh Harjani, and Antonio Maña. A security-focused engineering process for systems of embedded components. In *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systemss*, D4RCES '11, pages 4:1–4:9, New York, NY, USA, 2011. ACM.

[Roe01] Schumacher M. Roedig U. Security Engineering with Patterns. In *Pattern Languages of Programs*, 2001.

[SF] *Design of Secure and Energy-efficient Embedded Systems for Future Internet Applications (SECFUTUR), IST-25668, Seventh Framework Programme*. www.secfutur.eu.

[SSS11] Jayraj Signh, Arunesh Singh, and Ms. Raj Shree. Security Patterns in mobile Ad hoc Network: Requirement and Security Management Perspective. 2011.

[Ste06] et al. Steel C. *Core Security Patterns*. Pearson Ed. Inc., 2006.

[SYHJ08] Riccardo Scandariato, Koen Yskout, Thomas Heyman, and Wouter Joosen. Architecting Software with Security Patterns. 2008.

Combining Safety Engineering and Product Line Engineering

Jean-Pascal Schwinn[1], Rasmus Adler[2], Sören Kemmann[2]

[1]CT RTC SYE DAM-DE
Siemens AG
Otto-Hahn-Ring 6
81739 München
jean-pascal.schwinn@siemens.com

[2]embedded systems quality assurance
Fraunhofer IESE
Fraunhoferplatz 1
67663 Kaiserslautern
rasmus.adler@iese.fraunhofer.de
sören.kemmann@iese.fhg.de

Abstract: Product line engineering and safety engineering for software address current challenges in the development of software-intensive, safety-critical embedded systems. The two engineering disciplines have different goals and the approaches for achieving these goals have been created independently from each other. For this reason traditional safety engineering methods do not fit to traditional methods for software product line engineering. The research project "Safe ReSA (Safe Reusable Safety Artifacts)" between the Fraunhofer IESE and Siemens AG has the goal to extend traditional safety engineering methods so that safety engineering can be applied to the reusable artifacts that are created in product line engineering. Sequentially, we present how we extended methods for analyzing cause-effect relation between failures, for developing a safety concept and a safety case. Additionally, we present lessons learned from industry projects and our tool for applying the extended methods to complex real world systems.

1 Introduction

Product Lines Engineering (PLE) is an important development paradigm allowing companies to realize order-of-magnitude improvements in time to market and other business drivers. All PLE approaches have at least three things in common. First, they try to maximize the benefits of reuse. Second, they try to support reuse by separating and modularizing development artifacts. Third, they use model-based methods in order to achieve a certain degree of formality in the modularization of development artifacts.

Safety PLE focuses on reuse and modularization of artifacts from safety engineering. The safety artifacts include safety analysis results, the safety concept and the safety case. Safety analysis results describe cause-effect relations between failures. They are the input for deriving a safety concept because the safety concept explains all safety measures for breaking cause-effect relations between failures. The safety concept is in turn the input for the safety case, i.e., a defendable argumentation why the system is safe. Due to the dependencies between the three types of safety artifacts, we started with the modularization of safety analyses. In a next step, we integrated the modular safety analyses into development artifacts in order to apply them in a PLE context. Afterwards, we modularized safety concepts in order to support reuse. In the ideal case, it is possible to reuse some "safety element out of context"-safety concepts in order to build a new safety concept. The modularization of the safety concept is the basis for the modularization of the safety case because the safety case is an extension of the safety concept. It extends the safety concept with some evidences for the correct implementation of safety measures. In order to get from a modular safety concept to a modular safety case, it is necessary to formalize evidences and to attach them to the safety concept. The formalization and modularization of safety cases is the basis for checking automatically whether a composition of components is safe in a certain context. For a system with a predefined limited context this could enable a generation of a safety case at design time. Further, it is a powerful means for assuring safety in spite of openness. New technologies enable Car2Car communication or other things that make it impossible to safeguard all possible interaction scenarios at design time. A formal modular safety case could generate at runtime safety assurances according to the concrete runtime situation. However, "safety element out of context"-safety concepts, modular safety cases and safety cases for open adaptive system belong so far to challenges of future work.

This article presents results from our project "Safe Reusable Safety Analysis and Arguments" ("Safe ReSA") which aims at a holistic approach for safety PLE. It explains the evolution of our safety PLE approach and our lessons learned from applying the approach in industry projects. It is structured as follows. First, it presents the modularization of the fault tree analysis technique. Second, it explains how we integrated the modularized fault trees into system models in order to apply the modular fault tree analysis in a PLE context. Third, it presents how we formalized and modularized safety concepts. Based on this, it discusses challenges concerning the formalization and modularization of safety cases. Finally, it concludes with our lessons learned from applying safety PLE in industry projects and summarizes the benefits from our work.

2 Modularization of fault tree analysis

A popular safety analysis technique in safety engineering is Fault Tree Analysis (FTA). The modularization of FTA started in 2003 [KLM03]. The modularized fault trees are called component fault trees (CFTs). A CFT describes the relation between some output events and some input events. The relation is modeled in the same way as a normal fault

tree. CFTs can be composed by connecting the output events of one CFT with the input events of another CFT.

For implementing the CFT approach, we choose the UML-tool “MagicDraw” (MD, see [Ma13]) as modeling front end. MagicDraw supports the definition of domain specific modeling languages by adapting typical Unified Modeling Languages. We choose MagicDraw, because it is easy to change the modeling approach if it turns out that it does not fit to the capabilities of the modelers. In [ADH10], it is described in more detail how we integrated CFTs in MagicDraw.

Figure 1 shows a screenshot of a CFT in MagicDraw. The black triangles are output events. The yellow triangles are input events. If one goes into the CFT, the internal view shows how the two output events are caused by the input events and some internal events which are not visible from the external view.

For evaluating CFTs, we implemented interfaces to different calculation engines. For instance, we implemented a backend to the Siemens’ proven-in-use calculation engine Zusim. This enables us to apply novel modeling concepts in industry projects. Additionally, we implemented an own calculation backend in order to add calculations. For instance, we added calculations for weighing minimal cutsets according to their criticality, for considering the systems’ lifetime, for handling loops in models and for differentiating between several instances of one component and repeated elements. The enhancement of calculations was supported by the flexibility of MagicDraw, because it was easy to adapt the modeling language in cases where additional calculation input was required.

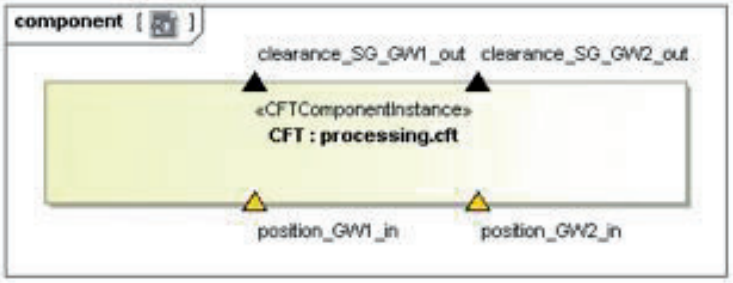


Figure 1: FTA component

3 Application of modular fault tree analysis in product line engineering

In industrial projects, we observed that modelers structured CFTs according to the system. A CFT typically referred to a function, a hardware component or a software component. For instance, the CFT in Figure 1 refers to the function *processing* in Figure 2. The two inputs events of the CFT refer to failure modes of the input signal

fct_proc_in and the two outputs events refer to failure modes of the output signal *fct_proc_out*.

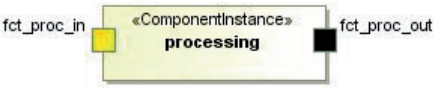


Figure 2: Functional component

The relation between a CFT and development artifacts helps to understand the CFTs and to keep them consistent to the system model. However, we observed that it was effort-intensive to keep the CFTs consistent to the system model. For this reason we integrated CFTs into the blocks of a block definition diagram. The blocks have input ports and output ports. As illustrated in Figure 3, the integration of CFTs was achieved by connecting every input event to exactly one input port and by connecting every output event to exactly one output port. Every event describes a failure mode of the signal that is exchanged via the port. The semantics of the signals depend on the semantics of the blocks. The blocks can represent functions, software components, or hardware components.

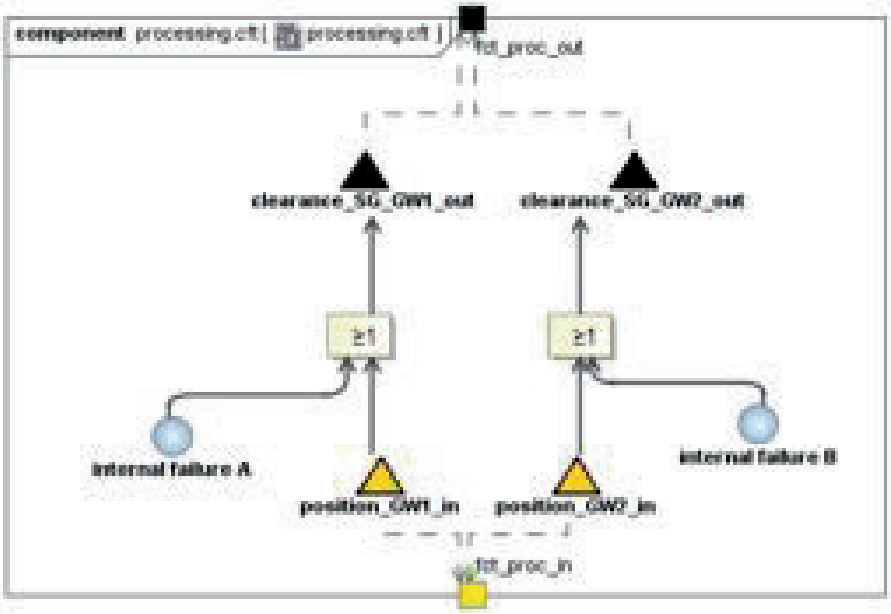


Figure 3: CFT of the functional

If blocks refer to functions then the block definition diagram explains how a composition of functions realizes some “features”, i.e., user-perceivable pieces of functionality. Features are an important aspect of PLE, because a product in a product line is often

considered a set of features. In order to apply the Component-integrated Component Fault Trees (C^2FT) in a PLE context, we developed a C^2FT modeling method for modeling the cause-effect relation between failures of functions and failures of features [AKS12].

We applied the C^2FT approach in many different industry projects. The integration reduced the modeling effort because the information about the system structure does not have to be modeled. Once the components are defined, analysis about the propagation of failures can be based on the interfaces defined in the functional model. This method may not only present interactions that would not have been obvious without the functional model, but can also help identify assumptions of failure propagation that are incorrect since for example the needed interface to enable this kind of failure is not existing according to the model. The result can be a highly complex fault tree providing a rather detailed representation of the analyzed system. Because of the different layers of representations of the system, the complexity becomes easier to manage than if only a single fault tree was used. For this reason, the integration of CFTs contributes to the comprehension of the cause-effect relation between failures and to the consistency between the cause-effect relation and the system models.

4 Modularization of safety concepts

After solving safety PLE issues concerning FTA, we focused in the project SafeReSA on the concept of safety validation. Safety validation covers all aspects of the safety process. It starts with the definition of safety goals and ends with the verification showing that the goals are met. In addition, not only technical aspects are considered, but also the process of quality and safety management itself has to be documented. Usually, the documentation of the safety validation is realized with Word files and Excel lists. Plain text is suitable for arguing the appropriateness of process quality and safety management. It is, however, very complex to explain the implementation of the safety goals with plain text. For coping with this complexity, one can use the Goal Structuring Notation (GSN, see [ACC11]). The GSN makes it possible to model how safety goals are iteratively broken down to safety sub-goals. However, the GSN provides no means for modularizing a safety concept and for linking the modular safety concept parts to reusable elements of PLE.

The idea in SafeReSA was thus to adapt GSN in order to model modularly safety concepts of reusable elements. For this purpose we developed first a GSN-like notation called “Safety Concept Tree” (SCT) [DFK09]. The root of a SCT refers to a safety goal. The SCT describes how this top-level safety requirement is refined by other safety requirements. A refinement is modeled with two simple gates. The first one is the AND-gate. When a requirement A is refined with an AND-gate into a requirement B and a requirement C, this means that requirement A is fulfilled when requirements B and C are fulfilled. The second gate is called decomposition gate (D-gate). When a requirement A is refined with a D-gate into a requirement B and a requirement C, this means that requirement A is still fulfilled, even if either requirement B or requirement C is not fulfilled. The gate is called decomposition gate as it is used to model the decomposition

of integrity levels. The idea behind the decomposition is to implement both refined requirements in order to create redundancy. As a consequence, the decomposed, redundant requirements can be implemented with a lower integrity level. In order to modularize a SCT, we introduced Safety Concept Parts (SCP). A SCP comprises guaranteed safety requirements (or simply guarantees), demanded safety requirements (or simply demands) and some Boolean gates for explaining the relation between guarantees and demands. Guarantees and demands of a SCP can refer to guarantees and demands of a component or a function. In this case it is desirable to have a formal link between the SCP and the development artifact. For this reason, we integrated the SCP into the blocks of a block definition diagram.

Figure 4 illustrates the SCP of the function *processing* in Figure 2. The SCP has two guarantees assuring that the output signal *fct_proc_out* has none of the two failure modes that are described in the CFT in Figure 1. The reason why the two guarantees are not collapsed to one guarantee is that they are provided with different integrity levels. As each failure mode of the output signal *fct_proc_out* can be caused by another failure mode of the input signal *fct_proc_in*, the SCT has two demands assuring that the input signal *fct_proc_out* has none of the two failure modes.

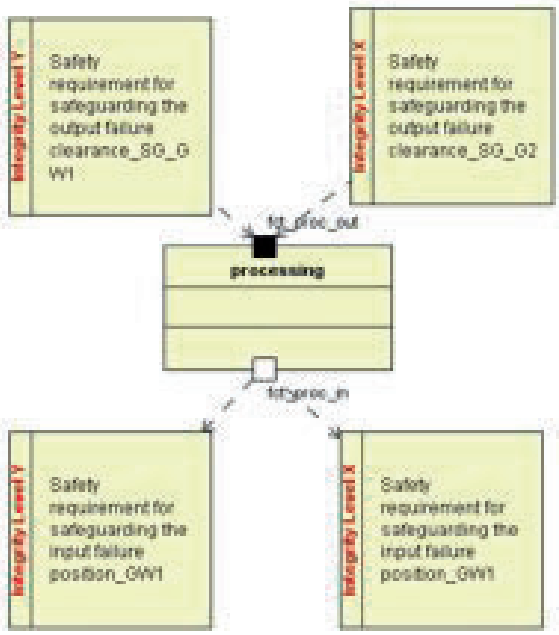


Figure 4: Component representation in safety concept view

We evaluated the modularized and component-integration safety concepts in several industry projects and observed several benefits. First, the modularization and component-integration contributes thus to the comprehensiveness of the safety concept. Even someone, who was not involved in the process of developing the safety validation

at all, can capture its content and reasoning more quickly visually than by having to read through many pages of several documents. A second benefit is that it is easier to keep the safety concept consistent to the system models. Consistency is obviously supported by the component-integration because some inconsistencies are avoided by the modeling language. For instance, whenever a block is removed from the diagram its related SCT is automatically removed. However, even the normal SCTs which are not component-integrated contribute to consistency, because any kind of documentation can be referenced from a SCT (e.g. via a hyperlink). If someone works with a document or creates a new one, it can be linked to the SCT and everyone else will work with the linked document. Since there should be only one link, confusion about different versions of the same document can be greatly reduced. The next user working with the model will automatically use the latest link, without the need to make sure first, that he is working on the latest version of the document he expects. Since files can be linked to the SCT using hyperlinks, any type of data can be attached.

The mentioned benefit for better comprehension is of course not only true for people building the safety concept, but also for the experts that need to assess it (e.g. independent assessors, notified bodies, etc.). Specific questions can be answered directly without flipping through many pages of documentation. In addition, once a certain part of the validation is completed, it can already be assessed, even if the whole analysis is still lacking some inputs. Whether this different approach will be feasible will also depend on the assessor and his commitment to perform modular assessments.

5 Challenges

Our modularization and component-integration of fault trees and safety concepts provide a good basis for safety PLE. However, there are still some challenges that have to be resolved for developing a holistic safety PLE approach from our model-based modularization and component-integration of fault trees and safety concepts.

The first challenge is to optimize reuse of component-integrated component fault trees and component-integrated component safety concept trees. The difficulty of reuse arises from the fact that safety is a property of the overall system. If a component is reused for realizing a new feature of the system, it is not self-evident that its related fault tree and its related safety concept can be reused. In particular the reuse of the safety concept is strongly limited because safety requirements are more detailed than failure modes. For instance, a common failure mode for a signal is *too late*. The cause-effect relation between *too late* failures of input and output signals has reuse potential as it is not defined how much too late. Considering a safety requirement for safeguarding a *too late* failure, it is necessary to define precisely in which time frame the signal has to be provided by the component. In case that the guaranteed time frame is quite large, it is likely that the guarantee is not sufficient in a new system context. Consequently, the safety concept cannot be reused. It is often very complex to check which safety models can be reused and which have to be adapted or totally rebuilt. This complexity increases if reasoning is tied to certain constraints, which need to be fulfilled to ensure the consistency of the safety argumentation. The obvious procedure would be to take the

component with its defined interfaces from the original safety validation, to transfer it into the new analysis and then to investigate, how the constraints fit in with the new validation. Depending on the amount of constraints, checking each of them one by one can lead to a rather high effort. This can result in a highly ineffective procedure since the outcome may be that none of them are relevant anymore. One approach to address this issue would be, to only allow a set of fixed interfaces, which are mandatory to all components used. While this would certainly ease integrating reused components into new analyses, it would at the same time decrease flexibility significantly and therefore limit new approaches of application.

The second challenge concerns the transition from a modular component-integrated reusable safety concept to a modular component-integrated reusable safety case. In the ideal case it is possible to extend a safety concept with evidences and to generate a safety case from the extended safety concept. However, it is even a challenge to come up with a semi-automated approach where only parts of the safety case are generated. Complete automation is the basis for the big future challenge to generate safety cases at runtime. This future challenge has to be solved in order to deal with the ever increasing openness of systems. Openness due to things like car2car-communication makes it impossible to consider all possible runtime situations at design time. Consequently, safety assurance has to be shifted from design time to runtime. The generation of safety cases at design time is a first step in this direction.

6 Conclusion

Addressing safety cases using a model based approach is a well established research topic, which has led to several publications in recent years (see e.g. [ADH10], [DFK09], [AKS12]). The following chapters describe some examples of new challenges faced and benefits seen putting the academic approaches into practical use.

6.1 Lessons learned

Putting the model driven safety case to practical use gave us additional input into the matter, raising new challenges:

Necessity for precise definition of safety goals: Fairly quickly it became clear that imprecise safety goals will complicate the modular approach for a safety case. The main reason for this is that it is often almost impossible to derive sub safety goals from a safety goal that does not precisely state conditions to be fulfilled. On the other hand, practical experience has shown that often safety goals tend to be formulated in a generic way, which often complicates proving its fulfillment. A paper addressing the aspect of the quality of safety goals is soon to be published. This also leads to the next observation:

Quality and readability of argumentation depends strongly on experience of the safety responsible: If the safety responsible has broad knowledge in the field of safety cases

and years of experience to fully compose a safety case, Safe ReSA can be a good support to document the safe development. However, if the responsible is rather new to field of safety, he may finally come to the same conclusion, but it may take him longer to get there. However, the first experiences have also shown that although a model based approach for a safety case is a rather new concept, the learning curve is quite steep.

6.2 Benefits

Putting the tooling into practical use had immediate positive effect on one of the first projects, where it was applied:

Use of different calculation engines: The possibility to use different calculation engines enabled the possibility to verify the results of the analysis. This added greatly to the confidence in this new approach. Additionally it enabled safety experts to switch to the new tooling without the need to switch to a new, unknown and not yet certified calculation engine.

Systematic identification of needless measures: Due to the formal approach of modeling functionality, safety concept trees and CFTs, already planned measures could be identified, which were not needed to reach the safety goal and could get removed. Since these measures would have meant implementation of software on two different subsystems with intricate communication between the two of them, a significant development effort could be saved.

High acceptance with customers: All customers, who got in contact with the graphical representation of the safety cases composed for them embraced the new approach. According to them, grasping even complex matters by seeing them in a graphical display made understanding them much easier, in contrast to having to read the same information in textual form.

References

[KLM03] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Mäckel. 2003. A new component concept for fault trees. In *Proceedings of the 8th Australian workshop on Safety critical systems and software - Volume 33* (SCS '03), Peter Lindsay and Tony Cant (Eds.), Vol. 33. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 37-46.

[ADH10] Adler, R.; Domis, D.; Höfig, K.; Kemmann, S.; Kuhn, T.; Schwinn, J.-P. and Trapp, M.: Integration of Component Fault Trees into the UML, In: Dingel, J.: Models in software engineering. Workshops and symposia at MODELS 2010. Reports and revised selected papers: Olso, Norway, October 3-8, 2010, pp 312-327.

[Ma13] Magicdraw homepage. URL: http://www.nomagic.com/. Last accessed on 2013/01/24.

[ACC11] Attwood, K.; Chinneck, P.; Clarke, M.; et al.: GSN COMMUNITY STANDARD VERSION 1, Origin Consulting (York) Limited, Nov. 2011.

[DFK09] Domis, D.; Förster, M.; Kemmann, S.; Trapp, M.: Safety Concept Trees, In: IEEE, Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual, 26-29 Jan. 2009, pp 212-217.

[AKS12] Adler, R.; Kemmann, S.; Schwinn, P. and Liggesmeyer, P.: Model-based Development of a Safety Concept, In proceedings 11th International Probabilistic Safety Assessment and Management Conference and the Annual European Safety and Reliability Conference 2012, Helsinki, Finland, 25-29 June 2012, pp. 4200-4208.

Doktorandensymposium

Vorwort zum Doktorandensymposium 2013

H. Lichter
RWTH Aachen University
lichter@swc.rwth-aachen.de

K. Schneider
Leibniz Universität Hannover
kurt.schneider@inf.uni-hannover.de

Eine Promotion ist in aller Regel eine Einzelleistung, auch wenn man mit vielen Kollegen zusammen in einem Forschungsprojekt arbeitet. Fragen, die sich Promovierende stellen und auf die sie Antwort suchen, sind beispielsweise: Wann weiß ich endlich genau, was das Thema ist? Komme ich zügig genug voran? Sind meine erzielten Ergebnisse schon ausreichend für eine Promotion? Natürlich ist dafür der betreuende Professor oder die Professorin der wichtigste Ansprechpartner.

Ein Doktorandensymposium bietet in diesem Rahmen die Gelegenheit, die eigene Situation zu reflektieren, vor anderen den Stand der Arbeit zu präsentieren und – nicht nur bezogen auf die Inhalte, sondern auch auf die Darstellung und die Herangehensweise – Rückmeldungen von anderen Promovierenden zu bekommen, denen es oft ganz ähnlich geht.

Doch damit nicht genug: Andere Professoren aus dem eigenen Forschungsbereich beurteilen die eingereichten Forschungsskizzen und kommentieren sie. Sie geben Tipps und Hinweise, die die Kandidaten später berücksichtigen können. Diese zusätzliche professorale Perspektive hat den Vorteil, dass die Relevanz des Themas und die Klarheit der eigenen Darstellung nicht nur von eigenen Kollegen betrachtet und bewertet wird, die mit dem Thema oft sehr gut vertraut sind. Wir danken unseren Kolleginnen und Kollegen für die Bereitschaft, durch gründliche Reviews und hilfreiche Kommentare allen Einsendern wertvolle Rückmeldung zu geben.

Nicht alle Bewerber und Bewerberinnen konnten ins Symposium aufgenommen werden. Nur wenn ein Thema schon erkennbar, aber noch nicht abschließend bearbeitet ist, kann man vom Doktorandensymposium optimal profitieren. Und auch die anderen Teilnehmer profitieren, wenn das Niveau einigermaßen vergleichbar oder sehr unterschiedlich ist. Wir haben für das Doktorandensymposium der SE 2013 eine Reihe interessanter Beiträge erhalten. Die passendsten davon haben wir zu einem Vortrag eingeladen. Sie werden Rückmeldung erhalten, aber auch geben.

Wir hoffen, dass dieses Doktorandensymposium allen Teilnehmerinnen und Teilnehmern hilft, die eigene Selbsteinschätzung zu kontrollieren, und Impulse für die weiteren Schritte zur Promotion gibt.

Guiding Transaction Design through Architecture-Level Performance and Data Consistency Prediction

Philipp Merkle
Software Design and Quality Group
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
merkle@kit.edu

Abstract: Designing transactional software which operates not only in a timely fashion but also preserves data consistency is challenging. While it is easy to preserve data consistency by choosing a high isolation level, this can quickly become a performance bottleneck due to limited concurrency. Conversely, relaxing the isolation between concurrent transactions may lead to data inconsistencies. Solving this trade-off systematically requires quantitative knowledge on the relation between transaction performance and the likelihood of data consistency violations under a given isolation level. Architecture-level performance prediction is a promising approach to address the first half of this trade-off but often neglects the influence of transactions. The second half—data consistency—is not addressed at all by existing approaches. Therefore, we plan to integrate transaction modelling into the Palladio approach for component-based software quality prediction. This creates the opportunity to predict not only performance metrics more accurately, but also to estimate data consistency violations.

1 Introduction

Transactional software is built to support use cases where data inconsistencies are intolerable or acceptable only to a certain extent. Transactions serve as a bundling mechanism for operations to be executed in an atomic, consistent, isolated and durable (ACID) fashion. Transaction isolation (the "I" in ACID) ensures that concurrent transactions do not affect each other concerning the data they access, or that at least the mutual influence is limited to a specified level—the so-called isolation level. For almost any relational DBMS, one can adjust the isolation level: globally for all transactions, per database session or even on a per-transaction basis.

Relaxing isolation is popular for optimising performance. Higher performance, however, comes often at the expense of data inconsistencies. This trade-off becomes apparent especially for the highest level of isolation: serialisability. Under this isolation level, concurrent transactions never interfere on a functional level. The database state resulting from executing two transactions t_1 and t_2 concurrently must have the same effect as executing them in a serial order; i.e. executing first t_1 followed by t_2 or vice versa. This inherently limits concurrency, mostly due to locks on shared data items. Serialisability, however, prevents all kinds of consistency violations due to concurrent data access including the well-known

dirty read and *lost update* anomalies. When choosing a relaxed isolation level, one must be well aware of potential data inconsistencies.

It is up to the software engineer to balance the transaction design between high performance and high consistency. For this, it is essential to understand how design alternatives influence performance and consistency. Design alternatives for transactions include transaction boundaries, database statements issued within these boundaries, the isolation level as well as the database schema.

Architecture-level software quality prediction is a promising approach to obtain quality metrics, on which design decisions can be based upon. For this, we plan to use the Palladio approach for component-based quality prediction [BKR09] as a foundation. Our goal is to integrate transaction modelling into the Palladio approach to obtain transaction quality metrics. These include throughput, abort and retry rates, as well as the degree of data consistency.

The scientific contribution of our research is a model-based approach for transaction quality analysis, integrated with architecture-level quality prediction. The envisioned approach includes a meta-model for transaction modelling, along with a corresponding model solver for predicting transaction quality metrics.

2 Research Goal and Questions

With our research, we intend to support software engineers in designing responsive yet consistency-preserving transactional information systems. We assume a component-based development process (cf. [KH06]) because it is well-suited for quality analyses [KBH07]. In our envisioned approach, component developers specify for each component its transactional behaviour—however, without having to provide each database statement in its full depth. Instead, a suitable abstraction is used. A model solver such as a simulator operates on these specifications to analyse how assembled components in a system influence each other. The solver yields predictions on the overall system performance and on the degree of data consistency.

The sketched approach raises various research questions, which are presented below. These questions mainly focus on performance influences of transactions while a detailed discussion of consistency is left for future work. Each research question comprises a motivation and a suggested solution. More details on the overall approach can be found in Sec. 3.

RQ1: What are the major performance factors of a transaction? For creating a suitable performance abstraction of transactions, it is vital to understand what factors influence a transaction's performance the most, and how they interact. We distinguish between factors stemming from a developer's design space, the deployer's configuration space, and performance factors due to concurrency. Design related factors are transaction boundaries, the complexity of encapsulated database statements and the isolation level under which these statements operate. Configuration related factors stem from the choice be-

tween different database management systems and from rich configuration options thereof. Concurrency factors include lock contention for database objects (e.g. tables, rows or indices) and contention for processing resources such as processors and storage devices. Contention is basically caused either by concurrent transactions or by competing software systems deployed along with the database on the same physical or logical machine.

Not all factors and interactions between them can be studied in the given timeframe, which makes a preselection inevitable. The selection process will favour design related factors since our goals in particular include early feedback on the quality of transaction design. Taking these factors as a starting point, we investigate their interrelation using two transaction processing benchmarks, TPC-W[1] and Apache Day Trader[2]. For automated and systematic experiments, we employ the SPA benchmark harness[3]. Initial experiments show how an increased abort rate due to serialisability leads to a significant drop in throughput.

RQ2: What is an appropriate abstraction level for modelling transactions? The purpose of this research question is to find a good balance between ease of modelling and prediction accuracy. A detailed model is hard to create in early development stages since many details are not known yet; it may, however, yield accurate predictions. By contrast, a highly abstract model is rather easy to create, e.g. from previous experience, but likely reduces prediction accuracy due to neglected factors. An example of a highly abstract model is the mere number of database statements issued by a single transaction; read versus write access is neglected as is the statements' complexity. An example for a detailed model is a full-fledged logical database schema along with a statistical data distribution per table column.

For specifying transaction boundaries and the isolation level, there is not much space for abstraction and we do not see a need to do so. Finding an appropriate abstraction for database statements is more challenging. We believe that probabilistic modelling may be well suited. For each database statement, the software engineer would characterise a number of parameters (e.g. read versus write access and read selectivity) using random variables. For example, one could specify that the customers table is read in 80% of the cases and written in the remaining cases; the selectivity of a read access could be set to 10%. An accompanying schema specification characterises the customers table in terms of its size. On this basis, a model solver could reason on the probability for a read-write conflict under a given concurrency level.

RQ3: How can transaction modelling be integrated with architecture description languages? As has been motivated before, we plan to base our approach on the existing Palladio approach. The Palladio approach includes the Palladio component model (PCM)—a meta-model for component-based software architectures—along with various analytical and simulative solvers capable of predicting the performance for instances of the PCM. Much research effort has been spent to make the PCM a conceptually clean language for

[1] http://www.tpc.org/tpcw/
[2] https://cwiki.apache.org/GMOxDOC20/daytrader.html
[3] http://sdqweb.ipd.kit.edu/wiki/Storage_Performance_Analyzer

architectural descriptions while being at the same time suited for different quality analyses. Preserving these properties throughout the process of integrating transaction modelling is challenging and is addressed by this research question.

Integrating transaction modelling means to extend the expressiveness of the PCM by introducing additional metaclasses on the metamodel level. For transaction demarcation, one could think of three additional actions BeginTX, CommitTX and AbortTX integrated into PCM's control flow abstraction. In between, a DatabaseStatement might issue a database demand similar to plain resource demands in the PCM. The database statement will likely be aligned with SQL statements but is in no case a full-fledged SQL statement.

RQ4: How can transaction-oriented quality prediction and architecture-level quality prediction be combined? Several analysis approaches exist for performance prediction based on a PCM model. Analytical approaches include layered queuing networks and queuing Petri nets. Simulative approaches include discrete-event simulators pursuing a generative [BKR09] or interpretive [MH11] approach. These solvers differ in expressiveness imposed by the underlying formalism, but also in maturity and coverage—some support just a subset of the PCM meta-model.

One of the main arguments for analytical approaches is their speed, which usually outperforms simulative approaches by magnitudes; however, at the cost of limited predictive power due to what is called the state space explosion problem. If one of the analytic approaches proves to be sufficiently powerful, we will stick to it for predicting transaction quality metrics. Otherwise, discrete-event simulation will serve this purpose.

3 Envisioned Approach

The artefacts involved in our approach and their interrelation are depicted in Fig. 1. The central component is the *transaction simulator*, which imitates a DBMS's transaction manager. Supplied with a transaction model and a DBMS profile, the transaction simulator predicts transaction quality metrics, including throughput, abort rates and the likelihood for various consistency violations.

The *transaction model* contains abstract behaviour characterisations for each transaction to be simulated. An example characterisation is the probability to access a certain database table, along with the total number of database accesses issued by that transaction. The transaction model references elements from the architecture model to allow for assigning transactional behaviour to certain architectural entities like a component specification.

The *DBMS profile* takes into account that transaction managers (TM) behave different, leading to different implications on performance and consistency. A locking-based TM, for instance, differs fundamentally from a TM based on multi-version concurrency control (MVCC); while the former acquires read locks in most isolation levels, the latter avoids read locks entirely. Even in the class of locking-based TMs, there are wide differences in the granularity of locks. While one TM features a sophisticated lock hierarchy, another

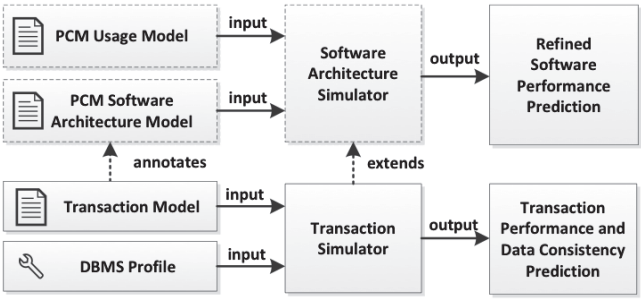


Figure 1: Artefacts in our approach and their interrelation. Dashed boxes indicate existing artefacts.

TM operates with fine-grained row-level locks only. This is why the transaction simulator is parametrised by a DBMS profile, which captures the characteristics of a DBMS. This way, the DBMS profile factors out product specifics from the transaction simulator.

To obtain system-level predictions, we integrate the transaction simulator into an existing but slightly modified *software architecture simulator*. For this, we use one of the simulators developed in the scope of the Palladio approach, e.g. EventSim [MH11], which has been specifically developed for extensibility. The architecture simulator expects two inputs: a PCM software architecture model, whose expressiveness is enhanced by the transaction model, and a corresponding PCM usage model. The *software architecture model* specifies the components and how they are assembled and deployed; the *usage model* specifies common use cases (called usage profiles), i.e. the workload. Whenever the architecture simulator encounters a transaction, it delegates the responsibility to the transaction simulator. This way, the usage profile propagates through the architecture down to transactions, where the transaction simulator keeps track of transaction-induced contention such as lock contention.

4 Related Work

A general overview of software performance evaluation (SPE) is provided in [BDMIS04]. SPE for component-based systems is surveyed in [Koz10]. From [Koz10], it becomes apparent that performance prediction approaches with a focus on middleware (including transactions) often neglect the influence of business logic and vice versa. A notable exception is the work by Menascé and Gomaa [MG00], who predict performance of a transaction-intensive client/server system. Using their CLISSPE language, they create detailed models of transaction behaviour along with information on the database schema and the DBMS. The high level of detail, however, makes it difficult to use their approach in early development stages. Data consistency prediction is covered by a few publications, e.g. [FGA09]. However, we do not know of approaches aimed at predicting data consistency on the system level.

5 Conclusion

In this paper, we argued for representing transactional behaviour explicitly in approaches for software quality prediction. Such an integrated prediction creates the opportunity to predict not only performance metrics more accurately, but also provides an estimate of data consistency violations. These metrics are supposed to help software engineers in finding a suitable balance between performance and data consistency. We base our approach on the existing Palladio approach for component-based software quality prediction, which has been successfully applied in a number of case studies.

References

[BDMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295 – 310, may 2004.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[FGA09] Alan Fekete, Shirley N. Goldrei, and Jorge Pérez Asenjo. Quantifying isolation anomalies. *Proceedings of the VLDB Endowment*, 2(1):467–478, 2009.

[KBH07] Heiko Koziolek, Steffen Becker, and Jens Happe. Predicting the Performance of Component-based Software Architectures with different Usage Profiles. In *Proc. 3rd International Conference on the Quality of Software Architectures (QoSA'07)*, volume 4880 of *Lecture Notes in Computer Science*, pages 145–163. Springer-Verlag Berlin Heidelberg, July 2007.

[KH06] Heiko Koziolek and Jens Happe. A QoS Driven Development Process Model for Component-Based Software Systems. In *Proc. 9th Int. Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *Lecture Notes in Computer Science*, pages 336–343. Springer-Verlag Berlin Heidelberg, 2006.

[Koz10] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010. Special Issue on Software and Performance.

[MG00] Daniel A. Menascé and Hassan Gomaa. A method for design and performance modeling of client/server systems. *Software Engineering, IEEE Transactions on*, 26(11):1066–1085, 2000.

[MH11] Philipp Merkle and Jörg Henss. EventSim – An Event-driven Palladio Software Architecture Simulator. In *Proc. Palladio Days 2011*, Karlsruhe Reports in Informatics ; 2011,32, pages 15–22, Karlsruhe, 2011. KIT, Fakultät für Informatik.

Erweiterung von Domänenspezifischen Sprachen um benutzerdefinierte Werttypen

Christin Zahner

Universität Hamburg, Fachbereich Informatik
Vogt-Koelln-Str. 30
22527 Hamburg, Germany
christin.zahner@informatik.uni-hamburg.de

Abstract: Domänenspezifische Sprachen unterstützen die Modellierung von Konzepten einer bestimmten Domäne. Mit Blick auf die Formulierung von Ausdrücken beschränken sich textuelle DSLs allerdings häufig auf die Unterstützung von primitiven Typen und Aufzählungstypen. Fachliche Konzepte, wie Geldbeträge oder Postleitzahlen, die zeit- und zustandslos modellierbar wären, sind nur schwer in eine DSL zu integrieren. Dieser Beitrag stellt Ansätze vor, um die Ausdruckskraft von DSLs um benutzerdefinierte Werttypen zu erweitern.

1 Ausgangssituation

Domänenspezifische Sprachen (DSL) versprechen mit ihrer limitierten Ausdruckskraft und ihrem Fokus auf eine bestimmte Domäne Vorteile gegenüber allgemeinen Programmiersprachen wie z.B. Java. Dazu gehören eine höhere Produktivität in der Programmierung, geringere Wartungskosten und eine verbesserte Kommunikation mit Domänenexperten [MH+05, Fo10]. Die *Modellgetriebene Softwareentwicklung (MDSD)* verfolgt ein ähnliches Ziel. Mit Hilfe von Modellen wird versucht auf einer höheren Abstraktionsebene zu arbeiten, als dies im Vergleich zu allgemeinen Programmiersprachen möglich ist [SV+07, KT08]. Die Modelle dienen dabei nicht mehr nur der Dokumentation, sondern werden auch für Transformationen oder zur Code-Generierung eingesetzt. Grundlegende Idee ist der Umgang mit Modellen auf verschiedenen Metaebenen. Ein Metamodell dient dabei zur Spezifikation einer Modellierungssprache [Se03]. Um die Kompatibilität zwischen Metamodellen sicherzustellen, wird eine gemeinsame Metasprache verwendet, ein Meta-Metamodell. Diese Einteilung findet sich in ähnlicher Weise bei Programmiersprachen wieder [Be01], vgl. Abbildung 1. Metasprache ist hier bspw. die *Erweiterte Backus-Naur-Form (EBNF)* und die Grammatik einer Programmiersprache entspricht ihrem Metamodell.

Im MDSD-Umfeld werden typischerweise Metasprachen eingesetzt, die auf objektorientierten Konzepten basieren. Bekanntes Beispiel ist die Meta Object Facility (MOF)[1]. Nicht alle Konzepte einer Domäne müssen jedoch zustandsbehaftet als Objekt

[1] http://www.omg.org/spec/MOF/

modelliert werden. Beispielsweise ist der Geldbetrag „5,00 EUR“ in den meisten Anwendungsfällen unveränderbar und wird nicht erzeugt. Auch hängen Operationen, wie bspw. die Addition zweier Geldbeträge, von keinem Zustand ab, ändern ihn nicht und liefern stets dasselbe Ergebnis. Eine Abstraktion in Form von zeit- und zustandslosen Werten kann hier helfen, die Konzepte der Domäne besser abzubilden. Daher wird u.a. in [Ma82, Zü04, Ev03] eine konzeptionelle Trennung zwischen *Objekt- und Werttypen* gemacht.

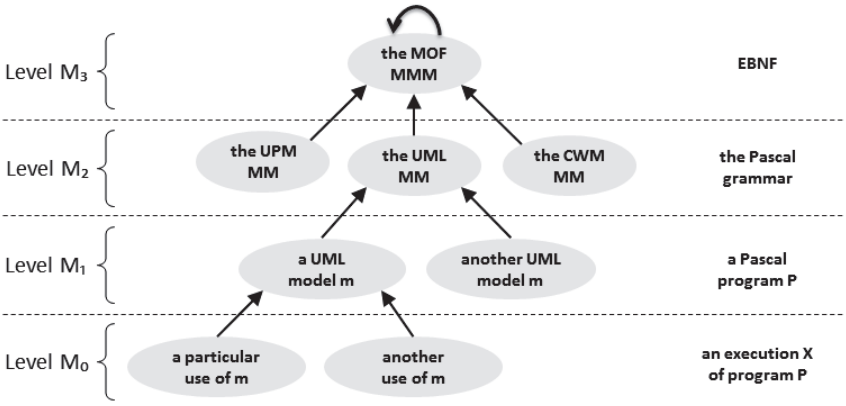


Abbildung 1: Metaebenen in der Modellierung und Programmierung am Beispiel der Modellierungssprache UML und der Programmiersprache Pascal („The OMG four layers standard modeling stack“ in [Be01])

2 Problemstellung

Sollen in einer DSL Berechnungen oder Bedingungen formulierbar sein, so muss der Sprachumfang Ausdrücke unterstützen. Für benutzerdefinierte Objekttypen ist die Verwendung in einer DSL auf ähnliche Weise denkbar, wie sie in objektorientierten Programmiersprachen erreicht wird: Syntaktisch sind allgemeine Sprachkonstrukte bspw. für Methodenaufrufe mittels Punktnotation vorstellbar. Im Rahmen der semantischen Analyse lässt sich zudem ein polymorphes Typsystem mit Konzepten wie Vererbung und dynamischem Binden implementieren. Für Werttypen ist diese Erweiterung um neue Typen ungleich schwieriger. Folgende Ansätze sind dafür vorstellbar:

a) Erweiterung der DSL-Werkzeuge (Lexer, Parser, Typsystem, etc.) mit jedem neuen Typ

b) Modellierung als Objekttyp

c) Abbildung auf primitive Typen (z.B. String, Boolean, Integer)

Variante a) verhindert die dynamische Erweiterung der DSL, da mit jedem neuen Werttyp die DSL-Werkzeuge um typspezifische Regeln u.a. zu Literalen, Operationen und Typumwandlungen erweitert werden müssen. Dieses Regelwerk wächst stetig und kann schnell sehr umfangreich und komplex werden, da es nicht von einzelnen Werttypen abstrahiert. Variante b) führt dazu, dass die wesentlichen Eigenschaften eines Werttyps nicht mehr durch die DSL sichergestellt werden können, z.B. seine Unveränderbarkeit. Zudem ist die Syntax sehr eingeschränkt. Variante c) wird häufig für DSLs gewählt. Jedoch geht dadurch die Fachlichkeit in der Formulierung von Ausdrücken einer DSL verloren. Der gewählte primitive Typ kann nicht die fachlichen Anforderungen an zulässige Literale und Operationen abbilden. Der arithmetische Ausdruck „5,00 * 0,19“ sagt nichts über die fachliche Bedeutung seiner Literale und Operationen aus. Es könnte sich um die Berechnung der Mehrwertsteuer handeln oder jede beliebige andere Berechnung. Die DSL bleibt in diesem Punkt eine allgemeine Programmiersprache.

3 Forschungsfragen

Ziel der vorgestellten Promotion ist es, die Ausdruckskraft textueller DSLs um benutzerdefinierte Werttypen zu erweitern, ohne dass eine Anpassung der DSL-Werkzeuge erfolgen muss. Die folgenden Fragen gilt es dafür zu beantworten.

1) Welche Informationen muss das Metamodell eines Werttyps enthalten, damit neue Werttypen modellierbar und in Ausdrücken einer DSL verwendbar sind?

2) Wie kann die Implementierung der syntaktischen und semantischen Analyse einer DSL von konkreten Werttypen abstrahieren? Welche allgemeinen Regeln für einen Werttyp können abgeleitet werden?

3) Wie lässt sich die Syntax einer DSL dynamisch um Literale und Operationen eines Werttyps erweitern?

4 Lösungsvorschläge

Zur Lösung der vorgestellten Problemstellung wird zunächst die Modellierung der Schnittstelle eines fachlichen Werttyps betrachtet. Um diese in einer DSL verwenden zu können wird dann die Integration auf semantischer Ebene untersucht. Abschließend wird die syntaktische Repräsentation eines Werttyps betrachtet.

4.1 Modellierung der Schnittstelle fachlicher Werttypen

Für die Modellierung eines neuen Werttyps ist ein passendes Metamodell notwendig, auf dem eine Modellierungssprache basieren kann. Mit Blick auf die bereits erwähnten objektorientierten Meta-Metamodelle gibt es dort eine Meta-Beschreibung sogenannter

„DataTypes“. Abbildung 2 zeigt einen Auszug aus der UML Infrastructure, die Basis der MOF ist. Mit objektorientierten Mitteln wird hier ein Werttyp samt seiner Eigenschaften modelliert. Ähnliche Ansätze existieren, um Werttypen in objektorientierten Programmiersprachen zu beschreiben, vgl. „Value Object“-Pattern in [Fo03, Ev03]. Auffallend ist, dass einige Eigenschaften von Werttypen hier nicht enthalten sind. Bspw. fehlen Subtypbeziehungen oder mögliche Literale eines nichtabzählbaren Werttyps. Der Fokus liegt auf primitiven Typen und Aufzählungstypen.

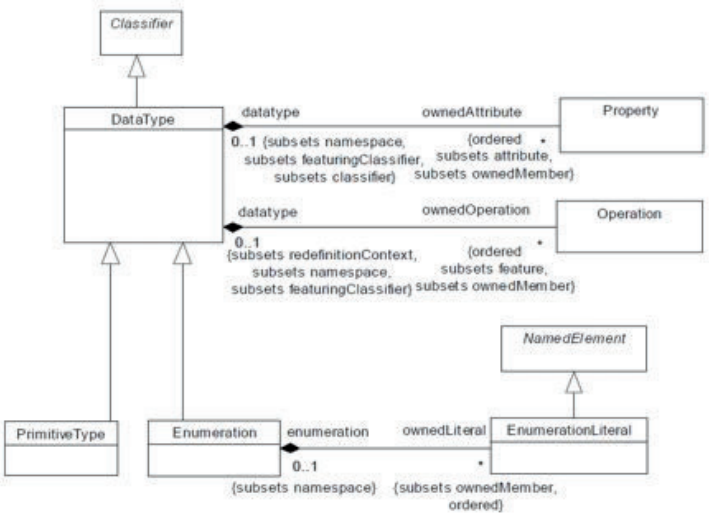


Abbildung 2: DataTypes in der UML Infrastructure [UML11]

Als Lösung wird eine entsprechende Erweiterung des Werttyp-Metamodells angestrebt. Das Meta-Metamodell der UML DataTypes dient dafür als Ausgangsbasis. Ziel ist es, die Schnittstelle eines Werttyps und damit seine abstrakte Syntax abzubilden.

4.2 Semantische Analyse: Integration von modellierten Werttypen in DSLs

Soll ein modellierter Werttyp nun in einer textuellen DSL verwendet werden, so müssen die zur Definition des Werttyps genutzten Modellierungskonzepte in die DSL integriert werden. In [KT08] werden verschiedene Integrationsansätze für Modellierungssprachen diskutiert. Für die vorliegende Problemstellung eignet sich insbesondere die Integration basierend auf einem gemeinsamen Metamodell.

Operationen eines Werttyps lassen sich bspw. in Form von binary methods [BL+95] modellieren. Einen ähnlichen Ansatz verfolgt u.a. Scala, indem Infixoperationen der Form „x + y“ als Methodenaufruf „x.+(y)“ interpretiert werden [OA+04]. Zusätzlich zum Metamodell einer Grammatik basiert die DSL somit auf dem Metamodel des Werttyps, um dessen statische Semantik zu prüfen. Die implementierten Regeln zu möglichen Operationen, zulässigen Operandentypen und Subtypbeziehungen können auf diese Weise von konkreten Werttypen abstrahieren.

4.3 Syntaktische Repräsentation und dynamische Grammatik

Für die syntaktische Repräsentation eines Werttyps muss seine konkrete Syntax innerhalb einer DSL definiert werden. Dies betrifft besonders die zulässigen Formate für Literale. Als Kernidee der Lösung dient hierfür der in [ES+95] beschriebene Ansatz einer Object-Oriented Language Definition. Eine Klassendefinition wird dort um die syntaktische Definition erweitert. Die Summe aller im aktuellen Kontext verwendbaren Objekte bildet dann eine dynamische Grammatik. Bezogen auf die Problemstellung der Werttypen muss also die Klassenbeschreibung des Wertttyps im Metamodell um syntaktische Definitionen erweitert werden. Dies kann bspw. in Form von Annotationen im Metamodell geschehen. Auf diese Weise ist auch die Verwendung des Metamodells in unterschiedlichen DSLs mit unterschiedlicher Syntax denkbar. Basierend auf den Annotationen kann anschließend ein Lexer und Parser generiert werden. Das Hinzufügen eines neuen Werttyps erweitert somit dynamisch auch die Grammatik der DSL.

5 Aktueller Stand der Arbeit

Die Arbeit erfolgt im Rahmen eines Drittmittelprojektes zur Realisierung eines Frameworks für die Modellgetriebenen Softwareentwicklung. Teil dieses Projektes ist die Neuentwicklung einer DSL für eine Regelsprache für Fachanwender. Diese textuelle DSL soll um Typen erweitert werden können, die sich mit Hilfe graphischer Notationen definieren lassen. Als technologische Grundlage kommen der Parser-Generator ANTLR[2] und Java zum Einsatz.

Die beschriebenen Lösungsvorschläge wurden in den vergangenen 2 Jahren im Rahmen der Implementierung dieser Regelsprache konkretisiert. Dafür wurde unter anderem ein Typsystem basierend auf den Informationen eines Werttyp-Metamodells implementiert. Hierbei wurde ebenfalls der Umgang mit primitiven Typen und speziellen Typen wie „null“ oder Sammlungen (Collections) untersucht. Die Lösungsansätze aus Abschnitt 4.1 und 4.2 sind somit in einer ersten Version umgesetzt. Aktuell noch ausstehend ist die Realisierung der dynamischen Erweiterbarkeit der DSL-Syntax. Als nächster Schritt steht hierfür die Formulierung von Annotationen am Werttyp-Metamodell an.

6 Forschungsmethodik, Vorgehen & Innovation

Um die geforderte Erweiterung der DSL um Werttypen zu erreichen, wird ein inkrementelles Vorgehen gewählt. Für diesen Prozess eignet sich insbesondere das Paradigma des *Design Reseach* [HM+04], dessen Fokus auf dem Entwurf und der Evaluation von Artefakten liegt. Dazu zählen Konzepte, Modelle, Methoden aber auch konkrete Realisierungen. Die beschriebene Methodik soll in mehreren Forschungszyklen umgesetzt werden. Hierfür werden Lösungen auf Basis der erwähnten Regelsprache erarbeitet. Die gewonnenen Ergebnisse sollen anschließend verallgemeinert, auf weitere Anwendungsfälle übertragen und somit evaluiert werden. Die Verallgemeinerung der

[2] http://www.antlr.org/

Lösung ist in Form einer allgemeinen Ausdruckssprache geplant, die in unterschiedlichen DSLs zum Einsatz kommen kann. Diese soll die Konzepte und Regeln für Werttypen zusammenfassend darstellen.

Die Innovation dieser Arbeit liegt auf zwei Ebenen. Einerseits wird der Benutzer einer DSL unterstützt, indem die Ausdruckskraft von DSLs verbessert wird. Fachliche Konzepte, die wertartig modellierbar sind, lassen sich dann mit einer konkreten Syntax in der DSL darstellen. Es muss nicht mehr auf primitive Typen zurückgegriffen werden. Dies ermöglicht auch eine bessere, fachliche Überprüfung von Ausdrücken auf Fehler. Außerdem wird diese Arbeit den DSL-Entwickler in Entwurf und Implementation von textuellen DSLs unterstützen. Sie wird Ansätze liefern, um modellierte Werttypen in die Werkzeuge einer DSL zu integrieren.

Literaturverzeichnis

[Be01] Bézivin, J.: From object composition to model transformation with the MDA. In Proceedings of TOOLS, Vol. 1, S. 350-354, 2001.

[BL+95] Bruce, K.; Leavens, G. T.; Castagna, G.; Cardelli, L.; Pierce, B.: On binary methods. In SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING: SYSTEMS, LANGUAGES, AND APPLICATIONS, ACM, S. 227-256. Harvard University Press, 1995.

[ES+95] Evered, M.; Schmolitzky, A.; Kölling, M.: A Flexible Object Invocation Language based on Object-Oriented Language Definition, 1995.

[Ev03] Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

[Fo03] Fowler, M.: Patterns of Enterprise Application Architecture (The Addison-Wesley Signature Series). Addison Wesley, 1 edition, 2003.

[Fo10] Fowler, M.: Domain-Specific Languages (Addison-Wesley Signature Series). Addison-Wesley Professional, 1 edition, 2010.

[HM+04] Hevner, A. R.; March, S. T.; Park, J.; Ram, S.: Design Science in Information Systems Research. MIS Quarterly, Volume 28, 2004.

[KT08] Kelly, S.; Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Pr, 3 2008.

[Ma82] MacLennan, B. J.: Values and objects in programming languages. SIGPLAN Not., 17(12): S. 70-79, 1982.

[MH+05] Mernik, M.; Heering, J.; Sloane, A. M.: When and how to develop domain-specific languages. ACM Comput. Surv., 37(4): S. 316-344, December 2005.

[OA+04] Odersky, M; Altherr, P.; Cremet, V.; Emir, B.; Maneth, S.; Micheloud, S.; Mihaylov, N.; Schinz, M.; Stenman, E.; Zenger, M.: An overview of the scala programming language. Technical report, Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[Se03] Seidewitz, E.: What models mean. Software, IEEE, 20(5): S. 26-32, Sept.-Oct. 2003.

[SV+07] Stahl, T.; Völter, M.; Efftinge, S.; Haase, A.: Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management Dpunkt Verlag, 2007.

[UML11] UML Infrastructure Specification, Version 2.4.1, August 2011, http://www.omg.org/spec/UML/2.4.1/, zuletzt aufgerufen: 20.11.2012.

[Zü04] Züllighoven, H.: Object-Oriented Construction Handbook: Developing Application-Oriented Software with the Tools & Materials Approach. Elsevier Science, 2004.

Multi-Language Refactoring with Dimensions of Semantics-Preservation

Hagen Schink
Institute of Technical and Business Information Systems
Otto-von-Guericke-University
Magdeburg, Germany
hagen.schink@gmail.com

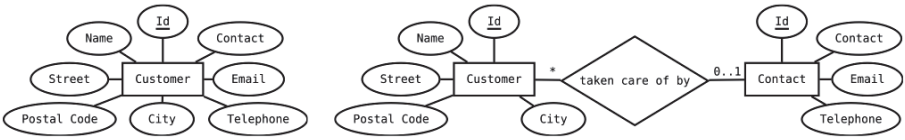
Abstract: Today, software developers utilize different *general-purpose* (GPL) and *domain-specific languages* (DSL) to implement *multi-language software applications* (MLSA). MLSAs, thus, contain artifacts of different GPLs and DSLs, e.g., source-code files and configurations. In a recent study we found that refactoring an artifact can break artifact interaction and that interaction cannot be re-established by additional refactorings. In this paper we propose an approach that supports developers in understanding and adapting changes to artifact interaction due to refactoring.

1 Introduction

Refactoring is a technique to modify a source-code's structure while preserving the source-code's semantics [Fow99]. Originally, refactorings are defined for single programming languages or paradigms. Thus, today refactorings exist for object-oriented and functional programming languages, and relational databases [Fow99, LT08, Amb03].

Today developers use multiple *general-purpose* (GPL) and domain-specific languages (DSL) in concert to implement software applications [SKL06, LLMM06, CJ08, Vis08, For08]. We call a software application implemented by means of different GPLs and DSLs *multi-language software application* (MLSA). An MLSA includes artifacts of different types, i.e., artifacts of different GPLs and DSLs. Developers access (interact with) different artifact types by means of *application programming interfaces* (API). We call a refactoring considering different artifact types of an MLSA and their interaction a *multi-language refactoring* (MLR). Current MLR implementations share the same idea: If a single-language refactoring breaks interaction with other artifact types, apply single-language refactorings to the interacting artifact types and, eventually, re-establish artifact interaction [SKL06, MS12]. But, in general, we cannot assume that suitable refactorings for all affected artifact types exist to re-establish the MLSAs semantics [SKSL11].

But what if no suitable single-language refactorings for interacting artifacts exist? Two options may be considered: (1) revert the initial single-language refactoring or (2) manually apply the necessary modifications to the interacting artifacts. These two options are dissatisfying because (1) a refactoring cannot be successfully applied or (2) manual modifications relying on a developer's intuition are required to complete the refactoring.



(a) Original table layout (b) Table layout after Split Table refactoring

Figure 1: The database schema (1a) before and (1b) after splitting the table `Customer`.

So, assuming that no suitable refactoring is available and a developer wants to complete a refactoring, how can we support the developer's refactoring effort in an MLSA?

In Section 2 we describe a use case for MLR. Section 3 describes a possible solution to improve the situation for developers applying MLRs. The current state of our work and our methodology is part of Section 4 before we present related work in Section 5 and conclude our work.

2 Motivating Example - Database Interaction

In this section we discuss the effect of the Split Table Refactoring [AS06, p. 145] upon C# code that accesses query results row by row using .Net's DataReader object. Therefor, we apply the Split Table Refactoring on the table `Customer`. Figure 1a shows table `Customer` which, originally, contains two different information: the customer's address and contact person. But certain information may be misleading, e.g. it is not obvious whether the attribute *`email`* belongs to the customer or the contact person. Thus, we split the table to separate the different information from each other. Figure 1b shows the resulting schema of the Split Table Refactoring. After refactoring the schema consists of the tables `Customer` and `Contact`. Table `Contact` holds all attributes related to a customer's contact person. A customer may only have at most one contact person and a contact person may takes care of zero or more customers. The C# application queries all customers and prints the customer's name and contact person details.

With a DataReader object we can access results of an SQL query row by row. Listing 1 outlines the basic usage of a DataReader. In Line 1 we create the DataReader object by executing an SQL statement. In the following Lines 3 to 6, we access the columns of each row of the result set by index.

After splitting table `Customer` the query in Listing 2 is no longer valid because the attributes *`contact`*, *`telephone`*, and *`email`* are, then, attributes of table `Contact`. At first sight this change is not obvious because the broken query will not become visible until runtime. At runtime an error is thrown indicating that certain columns referenced in the SQL query do not exist. When the developer becomes aware of the error, the developer is forced to understand the implications of the relational schema refactoring for the SQL statement in Listing 2 to be able to adapt the SQL statement accordingly. An option to adapt the SQL statement 2 is by defining an appropriate join as shown in Listing 3.

Listing 1: Reading results of query shown in Listing 2.

```
1 using (DbDataReader reader = cmd.ExecuteReader()) {
2   while (reader.Read()) {
3     string custName = reader.GetString(0);
4     string contName = reader.GetString(1);
5     string contTel = reader.GetString(2);
6     string conEmail = reader.GetString(3);
7   // data processing...
8 }
```

Listing 2: Query customer and contact information from the original table `Customer`.

```
1 SELECT name, contact, telephone, email FROM Customer
```

Listing 3: Query customer and contact information from the refactored table `Customer`.

```
1 SELECT name, contact, telephone, email FROM Customer
2   JOIN Contact ON (Customer.id_contact = Contact.id)
```

3 Improving MLR with Dimensions of Semantics-Preservation

Different APIs exist to interact with artifacts of different types. For instance, Section 2 shows one way of accessing a relational database from an object-oriented language. A different approach is to utilize an object-relational mapper [SKSL11]. So, the complexity of MLR involving relational database artifacts may range from rather simple SQL query modifications in Section 2 to complex modifications involving different artifact types. In general, we question the feasibility of tools for MLR as they exist for single-language refactoring because of the diversity of programming languages and APIs. But the question remains: How can we support developers in their MLR efforts?

Artifact interaction is realized through APIs, which in turn provide types and identifiers, e.g., data-structures and functions. Hence, a modification to types and identifiers may break artifact interaction. Our idea is to make these modifications to types and identifiers visible to developers by extracting and comparing type and identifier states. Type and identifier information are embedded in structures (e.g., tables, classes, procedures, and methods) of different artifact types (e.g., relational databases and object-oriented source-code). We call the different artifact types *dimensions of semantics-preservation* because different artifact types may consist of different syntactic structures supporting different refactorings and, thus, different kinds of semantics-preservaton. Then, it is a developers decision to choose refactorings and modifications to adapt interacting artifacts to the new type and identifier state.

Source code includes type and identifier expectations. In Listing 2, the query expects the columns *`name`*, *`contact`*, *`telephone`*, and *`email`* to be part of table `Customer`. In Listing 1, the code expects strings on the first four positions of the query result (Lines 3

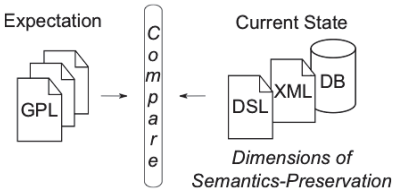


Figure 2: Comparison of types and identifiers extracted from interacting artifacts.

to 6). So, additionally, we may compare type and identifier expectations with the current type and identifier state, as Figure 2 shows, to determine if artifact interaction is broken.

Our idea is to visualize type and identifier states of artifact types in an IDE. Modern IDEs like Eclipse and Visual Studio already augment source code with a variety of information. In particular, developers get detailed information when types mismatch or identifiers cannot be resolved. But these information are only available for languages supported by the IDE at hand. Artifacts not supported by the IDE do not benefit from detailed type and identifier information. We argue that especially type and identifier information can help developers to identify issues after refactoring an MLSA.

4 Methodology and Current State

Basically, our work is based on three main hypotheses: (1) In general, it is not feasible to apply MLRs (semi-)automatically. (2) Only type and identifier information are necessary to understand artifact interaction. (3) Visible information about type and identifier modifications are sufficient to improve the usefulness of single-language refactoring in an MLSA. Based on these assumptions, we first search the literature for an appropriate model to describe and compare the type and identifier information of different artifact types. The next step is to implement front-ends to fill the model with type and identifier information from different artifacts. We plan to implement front-ends for Java and SQLite. Furthermore, we plan to integrate the model and front-ends into the Eclipse IDE as a prototypical plug-in. With the prototype we, then, conduct experiments to investigate the practicability and usefulness of our approach in regard to different use cases.

Currently, we investigate appropriate models to describe the type and identifier information. Furthermore, we started to get into the details of Eclipse Plug-In development.

5 Related Work

In the following, we introduce existing approaches to MLR and discuss their strengths and weaknesses.

The source-code meta-models FAMIX [Tic01], MOOSE [DLT00], and UML [VSMD03]

model object-oriented languages. The idea is to use FAMIX, MOOSE, as well as UML to generalize refactorings over a common meta-model. These approaches focus on object-oriented languages, and, thus, may not able to abstract artifacts of MLSAs in general. Another meta-model based approach is implemented in the IDE *X-Develop* upon a *Common Meta-Model* [SKL06]. The authors evaluate a refactoring in X-Develop on an MLSA. But the languages used in the MLSA can be compiled into a common base language, hence, the languages share common properties and, therefore, belong to the same artifact type in our understanding. Refactorings of other artifact types are not considered by the authors.

Some authors analyze and implement renaming for different artifact types [CJ08, KKKS08]. The authors show that MLR is possible for certain interactions (e.g. frameworks and the corresponding configuration files). In a recent study, we analyzed and implemented refactorings beyond renaming and showed that under certain conditions MLR is not easy to automate [SKSL11].

Coupled Software Transformations or *Co-transformations* are modifications of different interacting artifact types [Läm04]. Some argue that a semantics-preserving transformation of a database schema leads to transformations that do not modify the functionality of related applications [Cle09]. In a recent study, we applied both object-oriented and database refactorings [SKSL11]. Although we applied semantic-preserving transformations, i.e., refactorings on Java source-code and a relational database, we found cases where semantic-changing modifications are hardly avoidable.

In *model-driven architecture* (MDA), platform-specific models (PSM), e.g., classes in programming languages and database schemas in database instances, are generated from a high-level platform-independent model (PIM) [Ste08]. Our approach is different: First, we consider interaction between PSMs. Our approach does not depend on a PIM. Second, we consider refactorings in contrast to arbitrary transformations. Third, because we do not want to support automatic MLR, we may consider artifact-specific concepts if they influence interaction of artifact types.

6 Conclusion

The current idea of *multi-language refactoring* (MLR) requires that for each refactoring that affects artifact interaction a corresponding refactoring exists to re-establish a broken interaction. We argue that due to the plurality of available APIs this approach is hard to realize. We propose to extract type and identifier information of interacting artifact types and to make these information visible to developers, so developers are enabled to apply modifications themselves after a single-language refactoring broke artifact interaction.

References

[Amb03] Scott Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[AS06] Scott Ambler and Pramodkumar Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[CJ08] N. Chen and R. Johnson. Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support across Java and XML. *Workshop on Refactoring Tools*, pages 1–4, 2008.

[Cle09] Anthony Cleve. *Program Analysis and Transformation for Data-Intensive System Evolution*. PhD thesis, University of Namur, 2009.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. *CoSET*, 2000.

[For08] N. Ford. *The Productive Programmer*. O'Reilly, 2008.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[KKKS08] Martin Kempf, Reto Kleeb, Michael Klenk, and Peter Sommerlad. Cross Language Refactoring for Eclipse plug-ins. *OOPSLA*, 2008.

[LLMM06] Panos K. Linos, Whitney Lucas, Sig Myers, and Ezekiel Maier. A Metrics Tool for Multi-Language Software. *Undergraduate Research Conference*, 2006.

[Läm04] R. Lämmel. Coupled Software Transformations. *Workshop on Software Evolution Transformations*, 2004.

[LT08] Huiqing Li and Simon Thompson. Tool Support for Refactoring Functional Programs. *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, pages 199–203, 2008.

[MS12] Philip Mayer and Andreas Schroeder. Cross-Language Code Analysis and Refactoring. *SCAM*, pages 94–103, 2012.

[SKL06] Dennis Strein, Hans Kratz, and Welf Lowe. Cross-Language Program Analysis and Refactoring. *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 207–216, 2006.

[SKSL11] Hagen Schink, Martin Kuhlemann, Gunter Saake, and Ralf Lämmel. Hurdles in Multi-Language Refactoring of Hibernate Applications. In *Proceedings of the 6th International Conference on Software and Database Technologies*, pages 129–134. SciTePress - Science and and Technology Publications, 2011.

[Ste08] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software & Systems Modeling*, 9(1):7–20, December 2008.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Switzerland, 2001.

[Vis08] J Visser. Coupled Transformation of Schemas, Documents, Queries, and Constraints. *Electronic Notes in Theoretical Computer Science*, 200(3), 2008.

[VSMD03] P. Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards Automating Source-Consistent UML Refactorings. *UML*, 2003.

Generating Monitors for Usage Control

Frederik Deckwerth
Frederik.Deckwerth@es.tu-darmstadt.de *

Real-Time Systems Lab,
Technische Universität Darmstadt,
D-64283 Merckstraße 25, Darmstadt, Germany

Abstract: Protecting computational resources and digital information against unauthorized access is one of the fundamental security requirements in modern computer systems. Usage control addresses the control of computational resources after access has been granted. Despite its fundamental importance, no systematic methods exist to implement formal usage control specifications. This paper presents a model driven approach to solve this problem based on graph transformation. Using the precise semantics of graph transformation, access control models and policies can be formally analyzed in an early phase. The existing solutions on automated verification and efficient implementation of graph transformation show that this approach is suitable to address security concerns throughout the overall software development process.

1 Introduction and Motivation

Protecting computational resources and digital information against unauthorized access is one of the fundamental security requirements in modern computer systems. *Access control models* provide a formal foundation for expressing *access control rules* that specify which *subjects* (e.g., persons) are authorized to perform a specific *operation* (e.g., read or write) on a certain *object* (e.g., a computational resource). A set of access control rules intended to protect a specific system is called an *access control policy*. An *access decision* is the process of evaluating an access control policy to decide whether a given subject is authorized to perform a given operation on a given object.

Access decisions are typically made once for a request, and in case of a positive authorization decision, no ongoing control of the resource usage is performed. Nowadays, as digital information is exchanged in distributed computer systems, there is an increasing interest in controlling access beyond the initial access decision, which is addressed by the proposed usage control model (UCON) [PS04].

As in existing access control models, UCON utilizes attributes for access decisions that characterize subject and object properties. Addidionally, the main innovations of UCON are (i) mutable attributes and (ii) obligations. *Mutable attributes* can change their values as

*Supported by CASED (www.cased.de)

a result of access decisions or other not directly influenceable factors. As a consequence, authorizations can become invalid during an ongoing access, which requires the access decisions to be reevaluated every time an attribute changes. As an example consider policies that require the reduction of an account balance based on resource usage. *Obligations* specify mandatory tasks to be performed by a subject before (pre-obligation) or during (on-obligation) resource usage. While on-obligations have to be evaluated continuously during resource usage, pre-obligations require a "history function" to check if the mandatory activities where previously performed. For example, medical personnel must sign a privacy policy before reading patient records.

Most access control models (including UCON as defined in [PS04]) do not explicitly support access decisions based on structural relational dependencies between objects and subjects. Although such dependencies can be expressed by arbitrary valued attributes, making them explicit often facilitates the specification and implementation of access control policies. Consider, for example, a social network scenario where access decisions are mainly based on friendship relations. Expressing this as an attribute requires every user to maintain an attribute (e.g. a list) that stores the names (i.e. unique identifiers) of all friends, which has to be kept consistent. In this case, encoding "friendship" as a structural relation (e.g., as an association) can lead to simpler and more readable access control policies.

Mutable attributes, obligations and explicit structural constraints impose new requirements concerning the formalism for specifying and the techniques for implementing such systems. The formalism should capture the combination of temporal aspects introduced by the obligations as well as structural relation. The need for a continuous evaluation of attributes and structural constraints combined with temporal constraints requires non-trivial run-time monitoring techniques.

As a consequence, deploying access control subsystems in real-world applications requires a systematic approach to realize systems that behave as expected. From an end-user perspective, the following basic requirements can be identified:
Formal model of the access control policy. To faithfully capture and analyze access control policies, a theoretical framework is needed that (i) provides a set of high-level concepts to specify a policy and (ii) is amenable to formal analyzing and verifying desired properties.
Systematic implementation. In practical applications, the security of usage control system does not only depend on the correctness of the policy, but also on its correct implementation. To implement a policy, the high-level specification has to be translated into an executable implementation. The more systematic this translation, the easier it is to ensure that the properties proven for the policy model also hold for the implementation.
Seamless integration. As systems are typically not built from scratch, it must be possible to integrate access control modules into legacy systems. To this end, it must be possible to tailor the implementation to the needs of the system.

2 Related Work

Several formal languages [Mar07, PHB06] have been introduced to express UCON policies with temporal logic or process algebra. These approaches focus on the specification of the control flow (i.e., the sequence of actions required to gain access) and do not address complex structural constraints. On the other hand, there exist different implementations to enforce usage control [GNC10, WPH11]. Most of these approaches focus on a specific application domain such as service-oriented architectures or assume specific infrastructures. Other more general approaches offer implementations of access control decision modules [NPDG11, KZB+08], which can handle policies written in a specific language, but are difficult to integrate into legacy systems as they cannot be easily adopted. Although [KP12] investigates how to systematically translate high-level policy specifications to implementation specific policies, their implementation is based on a static policy decision module and requires a manual implementation of modules for managing operations and attributes. Complex structural constraints are also not supported.

3 Sketch of the Solution

I propose a model-driven approach based on graph transformation, which is a rule-based description for the manipulation of graph structures. Based on the precise semantics of graph transformation, access control models and policies can be formally analyzed in an early phase. The existing solutions on automated verification and on translation of graph transformation specifications into efficient implementations show that this approach is suitable to address security concerns throughout the overall software development process. Nevertheless, there are still open challenges regarding UCON. Existing approaches based on temporal logic or process algebra are suitable for expressing temporal constraints but lack support for expressing structural constraints. The situation for a graph transformation based approach is exactly the opposite: structural constraints can be easily expressed and verified, but temporal constraints are difficult to handle as graph transformation does not offer an explicit mechanism for the control flow. As none of the approaches is necessarily superior, I propose to investigate how to combine the strengths of both approaches. This leads to several challenges regarding the formalism as well as its efficient implementation.

3.1 Expressing Access Control Policies with Graph Transformation

The general viability of graph transformation to express and analyze access control models and policies is illustrated in [KMPP02]. The approach demonstrates that role based access control (RBAC) [SCFY96] with separation of duty constraints, which prohibit assigning two particular roles to the same subject, can be expressed by graph transformation. The basic idea is to model the system state by a graph called a *system model* and state changes by applying graph transformation rules to the system model. A graph transformation rule

consists of three graphs called the left-hand side (LHS), the right-hand side (RHS), and the negative application condition (NAC). The LHS together with a NAC defines the application condition of a rule. A rule is applied to the system model by (i) finding a match (i.e., a structural identical part) of the pattern described by the LHS, (ii) checking the negative application condition (which prohibits the presence of certain structures) and (iii) replacing the match in the system model by the RHS. Separation of duty constraints are mapped to *graph constraints* which is a graph that specifies a forbidden pattern. A system model is said to be in an allowed state if it does not contain a forbidden pattern. By proving that none of the rules can transform an allowed state into a forbidden one, it can be guaranteed that the system never evolves into a forbidden state, provided that the initial system model was an allowed state. Such a static analysis can be performed in an automated fashion. Moreover, the rules can be adopted automatically by adding NACs that prevent rules to be applied for exactly those situations that would lead to a forbidden system model state.

This powerful constructive approach is well understood for structural patterns [HW95]. As UCON strongly relies on non-structural constraints (e.g., constraints on integers or strings) it has to be investigated how the approach can be extended to also handle non-structural constraints. From a theoretical point of view, non-structural constraints can be treated as invariants and proven inductively. However, in general applications, such invariants are hard to check automatically. Based on the rule based character and explicit treatment of NACs, graph transformation might be feasible for such an automated analysis. Inspired by the treatment of non-structural constraints [AVS12], I consider an approach that can verify structural as well as non-structural constraints as feasible.

As previously discussed, obligations require temporal constraints to express before/after relations. The integration of temporal logic with graph transformation has been studied in [GHK00, RS06] from a theoretical point of view. However, regarding the implementation, it is still an open topic how to efficiently realize run-time monitors that can evaluate constraints combining structural and temporal expressions.

3.2 Implementation

Several graph transformation engines have been proposed to execute a formal graph transformation specification or to check constraints in a graph. All engines have to solve a common problem: the efficient querying of complex graph-based model structures, which can be expressed as an NP-complete graph pattern matching problem. In general, the solutions can be categorized into batch and incremental approaches. While batch algorithms start the search for a pattern for each request from scratch, incremental approaches explicitly store all matches and incrementally maintain these matches when the system model is updated. Using this approach, it is possible to obtain all matches of a pattern in constant time, which makes it ideally suited for monitoring graph structures. However, this is achieved at the expense of extra memory for maintaining the matches and additional time in the update phase.

During my previous work, I was involved in developing heuristics for improving the run-

time of an batch pattern matching process [VDWS12]. These techniques may be also applied to incremental approaches to reduce the time for updates and memory consumption.

Monitoring temporal constraints requires a "history function" to check if an activity was performed in the past. As access control systems usually run for long periods, logging all activities is not feasible. Several algorithms [BKM10, MJG+12] to efficiently maintain such "history functions" exists, however, a combination of these approaches with incremental pattern matching is still an open topic.

4 Plan of Action

I plan to implement a framework based on graph transformation for (i) modeling usage control policies (ii) formally analyzing usage control policies and (iii) generating efficient monitors for usage control, which requires appropriate extensions of the pattern matching engine that is currently under development at our group [VAS12]. More specifically:

Modeling usage control policies. In a first step it has to be investigated how the usage control model can be formalized using graph transformation. This includes how to exactly identify the concepts that are difficult to express by graph transformation. Based on these results, an appropriate extension to graph transformation can be developed.

Analyze and formal verify access control policies. It has to be investigated to what extent constraints that combine structural and non-structural restrictions can be verified statically. Moreover, it has to be investigated whether this is also possible for temporal constraints. Constraints that cannot be statically verified have to be checked at runtime by monitoring techniques.

Generate efficient monitors for usage control. In the first step, the pattern matching engine currently developed at our group has to be extended to support incremental pattern matching. In a next step, it has to be investigated how temporal aspects can be integrated.

Finally, the viability of the approach has to be evaluated based on a non-trivial case study. Additionally, I plan to develop a benchmark to provide a basis for a performance comparison with other approaches.

References

[AVS12] A. Anjorin, G. Varró, and A. Schürr. Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. *Electronic Commun. of the EASST*, 49, 2012.

[BKM10] D. Basin, F. Klaedtke, and S. Müller. Monitoring security policies with metric first-order temporal logic. In *Proc. of*, SACMAT '10, pages 23–34. ACM, 2010.

[GHK00] F. Gadducci, R. Heckel, and M. Koch. A Fully Abstract Model for Graph-Interpreted Temporal Logic. In Hartmut Ehrig, Gregor Engels, Hans-Jrg Kreowski, and Grze-

gorz Rozenberg, editors, *Proc. of TAGT '00*, volume 1764 of *LNCS*, pages 310–322. Springer, 2000.

[GNC10] G. Gheorghe, S. Neuhaus, and B. Crispo. xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In Masakatsu Nishigaki, Audun Jsang, Yuko Murayama, and Stephen Marsh, editors, *Trust Management IV*, volume 321 of *IFIP*, pages 63–78. Springer, 2010.

[HW95] R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars-a constructive approach. *ENTCS*, 2:118–126, 1995.

[KMPP02] M. Koch, L. V. Mancini, and F. Parisi-Presicce. A graph-based formalism for RBAC. *ACM Trans. Inf. Syst. Secur.*, 5(3):332–365, August 2002.

[KP12] P. Kumari and A. Pretschner. Deriving implementation-level policies for usage control enforcement. In Proc. of CODASPY '12, pages 83–94. ACM, 2012.

[KZB+08] B. Katt, X. Zhang, R. Breu, M. Hafner, and J. Seifert. A general obligation model and continuity: enhanced policy enforcement engine for usage control. In Proc. of SACMAT '08, pages 123–132. ACM, 2008.

[Mar07] F. Martinelli. A model for usage control in GRID systems. In *SecureComm '07*, page 520, 2007.

[MJG+12] PatrickOâNeil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rou. An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14:249–289, 2012.

[NPDG11] R. Neisse, A. Pretschner, and V. Di Giacomo. A Trustworthy Usage Control Enforcement Framework. In *Proc. ARES '11*, pages 230 –235, aug. 2011.

[PHB06] A. Pretschner, M. Hilty, and D. Basin. Distributed usage control. *Commun. ACM*, 49(9):39–44, 2006.

[PS04] J. Park and R.S. Sandhu. The UCONABC usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

[RS06] T. Rötschke and A. Schürr. Temporal Graph Queries to Support Software Evolution. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *ICGT '06*, volume 4178 of *LNCS*, pages 291–305. Springer, 2006.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *Computer*, 29(2):38 –47, 1996.

[VAS12] G. Varrø', A. Anjorin, and A. Schürr. Unification of Compiled and Interpreter-Based Pattern Matching Techniques. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Strrle, and Dimitris Kolovos, editors, *ECMFA '12*, volume 7349 of *LNCS*, pages 368–383. Springer, 2012.

[VDWS12] G. Varró, F. Deckwerth, M. Wieber, and A. Schürr. An Algorithm for Generating Model-Sensitive Search Plans for EMF Models. In Z. Hu and J. Lara, editors, *Proc. of ICMT*, volume 7307 of *LNCS*, pages 224–239. Springer, 2012.

[WPH11] A. Wahl, S. Pfister, and B. Hollunder. Complex Event Processing for Usage Control in Service Oriented Infrastructures. In *Proc. SERVICE COMPUTATION '11*, pages 92–97, 2011.

Eine Multikanal-Architektur für adaptive, webbasierte Frontendsysteme und deren Erweiterbarkeit durch Variantenbildung

Dipl.-Inform. Michael Thomas Hitz
DHBW-Stuttgart + Allianz Deutschland AG
Paulinenstraße 50
70178 Stuttgart
hitz@dhbw-stuttgart.de

Abstract: Der in den letzten Jahren stark gestiegene Bedarf an webbasierten Zugängen zu den Systemen eines Unternehmens für unterschiedliche Nutzergruppen (fachliche Kanäle) führte häufig dazu, dass parallel *monolithische*, *siloartige* Frontend-Systeme entstanden sind, welche in Wartung und Weiterentwicklung durch die resultierenden Redundanzen hohe Kosten verursachen. In den letzten Jahren rückten zudem neue Technologien wie mobile Geräte weiter in den Fokus, die neben den fachlichen Kanälen eigene Anforderungen an die Darstellung und die Prozesse haben und damit die Komplexität weiter erhöhen.

Um schnell und kosteneffizient auf den Markt reagieren zu können, bedarf es einer Lösung, welche Funktionalitäten kanalübergreifend wiederverwendbar macht und bei Änderungen oder kanalspezifischen Erweiterungen Redundanzen durch Bildung von Varianten vorhandener Oberflächen und Prozesse vermeidet.

1 Einleitung

1.1 Motivation

Die letzten Jahrzehnte der Softwareentwicklung im Web-Umfeld produzierten Frontend-Systeme[1], die spezifisch auf Nutzergruppen (**fachliche Kanäle**) zugeschnitten wurden. Diese Fokussierung der in den entstandenen Portalen entwickelten Anwendungen auf einen bestimmten Kanal führte in vielen Fällen dazu, dass Oberflächen und Anwendungsabläufe mehrfach entwickelt wurden, obwohl diese häufig für die fachlichen Kanäle gleich sind oder in nur leicht modifizierter Form wiederverwendbar wären.

Mit der stetig wachsenden Verbreitung neuer Technologien wie z.B. Spracherkennung und intelligenterer mobiler Geräte müssen weitere Anforderungen durch die Oberflächen und

[1] Der Begriff *(webbasiertes) Frontend-System* wird hier als Summe aller Komponenten verwendet, die notwendig sind, eine Anwendung dem Nutzer zu präsentieren. Dies beinhaltet Oberflächen, Seitenfluss und Anwendungslogik, welche den Zugriff auf Prozesse in den Backendsystemen orchestriert. Die Backendsysteme werden dabei als bestandsführend betrachtet und stellen die eigentlichen Businessfunktionalitäten bereit - und sind damit nicht Teil des Frontend-Systems.

Anwendungsabläufe erfüllt werden (z.B. darstellbare Datenmenge, eingeschränkte Konnektivität). Aus Systemsicht sind dies weitere Kanäle (**technische Kanäle**), für die bisher meist ebenfalls eigene Portale parallel aufgebaut wurden.

Die Entstehung solcher monolithischen *Entwicklungs-Silos* hat sowohl organisatorische als auch technische Ursachen ([Ban01]). Organisatorische Gründe sind dabei insbesondere die Organisationsstruktur der IT in größeren Unternehmen, die meist entlang der fachlichen Kanäle erfolgt und damit eine isolierte Softwareentwicklung begünstigt. Technische Ursachen liegen beispielsweise in der mangelnden Unterstützung des Szenarios durch die vorhandene Infrastruktur und Frameworks.

Die Redundanzen, die in der Systemlandschaft entstanden, sind nur schwer zu verwalten und verursachen in der Wartung und Erweiterung hohe Kosten. Eine Lösung ist die Schaffung eines multikanalfähigen Systems[2], in welchem Varianten zur Unterstützung neuer Kanäle schnell und zu geringen Kosten integriert werden können.

1.2 Ziel und Durchführung der Arbeit

Ziel der Arbeit wird es sein, einen Blueprint zu einer adaptiven, multikanalfähigen Frontend-Architektur und Vorgehensweisen zu erarbeiten, welche die folgenden Eigenschaften aufweisen:

- ein Basismodell, welches die Nutzung vorhandener Elemente über mehrere Kanäle hinweg vorsieht
- eine einfache und schnelle Integrierbarkeit neuer Kanäle (fachlich und technisch) und deren Spezifika durch Adaption und Variantenbildung von Oberflächen und Prozessen im Frontend gestattet

Hierzu werden anhand eines Szenarios aus dem Versicherungsumfeld Anforderungen und Eigenschaften abgeleitet, die ein Multikanalsystem aufweisen muss. Darauf aufbauend wird ein Basismodell für eine Architektur entwickelt, welches eine grundsätzliche Schichtung des Systems vorschlägt und über die Verantwortlichkeiten der Schichten eine detailliertere Betrachtung der Kernelemente gestattet.

Basierend hierauf fokussiert die Arbeit im Weiteren auf die Erforschung und das Design von Mechanismen, wie Variantenbildung auf Prozessebene erreicht und darauf aufbauend kanalspezifische graphische Oberflächen[3] (automatisch oder ebenfalls durch Variantenbildung bestehender Abläufe) abgeleitet werden können. Das Ziel ist hierbei, eine hohe Wiederverwendung durch Variantenbildung bestehender Anwendungen zu erreichen.

[2] Im Rahmen dieser Arbeit wird unter dem Begriff *Multikanalsystem* ein System verstanden, das in der Lage ist, unterschiedliche fachliche und technische Kanäle einheitlich zu bedienen, indem durch dessen Aufbau ein hoher Grad an Wiederverwendung erreicht wird und trotzdem die Spezifika der einzelnen Kanäle abgebildet werden können.

[3] Den Schwerpunkt der Untersuchungen bilden hierbei webbasierte Oberflächen. Ziel ist jedoch auch die Anwendbarkeit der Lösungen auf Desktop-Anwendungen, die hinsichtlich der Präsentation neben webbasierten Browser- und mobilen Anwendungen weitere Anforderungen stellen.

1.3 Forschungsfragen

Die daraus resultierenden Fragen für die weiteren Forschungen sind

- Welche Anforderungen leiten sich aus der Forderung nach *Multikanalfähigkeit* eines Systems ab und wie lassen sich diese Anforderungen in einem Basismodell einer Architektur erfüllen?
- Wie können Prozesse multikanalfähig definiert und durch Variantenbildung die Spezifika weiterer Kanäle berücksichtigt werden?
- Wie können kanalspezifische Oberflächen in einem Multikanalsystem für diese Prozesse erstellt werden, sodass schnell und kostengünstig Erweiterungen erfolgen können?

1.4 Methodik

Zur Analyse der Anforderungen an ein Multikanalsystem wird ein Szenario aus dem Versicherungsbereich gewählt, welches fachlich durch strukturierte Interviews mit einem oder mehrerer am Forschungsprojekt beteiligter Partner erstellt wird. Zudem erfolgt eine IST-Analyse auf Architekturebene, um daraus ggf. weitere, technische Anforderungen und Problemstellungen ableiten zu können.

Darauf aufbauend und unter Einbeziehung relevanter Forschungsergebnisse und Literatur in diesem Umfeld, wird ein Basismodell für eine Multikanalarchitektur hergeleitet. Dieses soll mittels geeigneter Vorgehensweisen hinsichtlich seiner Qualität bewertet und auf die Tragfähigkeit geprüft werden. Hierzu existieren in der Literatur unterschiedliche Ansätze, von denen ein geeignetes Verfahren ausgewählt wird (z.B. [KLKB07]).

Die Forschungen im Bereich der Oberflächen und Prozesse und technischen Herausforderungen gründen auf Forschungsergebnissen, welche Teilaspekte der Aufgabenstellung behandeln. Die Verifikation der im Rahmen des Projekts erarbeiteten Ergebnisse erfolgt durch prototypische Umsetzung bzw. realem Einsatz bei am Projekt beteiligter Partner (in Zusammenarbeit mit der DHBW Stuttgart, der Hochschule für Technik Stuttgart und der Allianz Deutschland AG).

2 Verwandte Arbeiten

Es existiert eine Reihe von Arbeiten, die architekturelle Aspekte (multikanalfähiger) Anwendungen untersuchen (z.B. [BGK+97], [Per06], [ZDGH05], [YN08]), zu Themenstellungen hybrider Oberflächentechnologien und zu Konzepten zur Lösung technischer Herausforderungen in diesem Umfeld (z.B. [BGL06],[JrTD04], [MP02]). Auf Ebene serviceorientierter Architekturen und Prozessmanagements existieren Forschungen im Bereich mandantenfähiger und adaptiver Prozesse (z.B. [WKNL07], [DR09]), die einen grundlegenden Beitrag leisten.

Die vorgeschlagene Arbeit soll die Forschungen im Bereich multikanalfähiger Systeme um den Themenbereich der Erweiterbarkeit durch Variantenbildung auf Oberflächen- und Prozessebene ergänzen und sie im Rahmen eines Multikanalsystems zusammenführen.

3 Erste Ergebnisse - Stand der Arbeit

Die Arbeit befindet sich in der Phase der Vorstudien, welche die Machbarkeit bereits vorhandener Lösungsideen für die oben angesprochenen Forschungsfragen untersuchen.

Grundlage hierfür waren erste Analysen im Versicherungskontext für Multikanalsysteme relevanter Nutzergruppen. Als erstes Szenario aus dem Versicherungsbereich wurde das Thema „Tarifierung mit Antrag“ gewählt, da es sich hier um geschlossene Prozesse handelt, die von mehreren Kanälen mit leichten Variationen durchgeführt werden.

3.1 Erstes Basismodell

Das hergeleitete Basismodell entstand unter der Annahme, dass mehrere fachliche Kanäle mit unterschiedlichen Geräten das webbasierte System nutzen. Das System soll für alle Kanäle einheitlich sein, sodass dieselbe Infrastruktur verwendet werden kann. Basierend auf den bisherigen Erfahrungen in der Portalentwicklung und der grundsätzlichen Entscheidung, einen serviceorientierten Ansatz bei der Integration der Backend-Systeme zu verwenden, ergab sich das in Abbildung 1 dargestellte Basismodell. Es folgt auf der Businesslogik-Ebene den in [BGK+97] angeführten Grundsätzen und konkretisiert die technischen Applikationsaspekte.

Den *kanalspezifischen Einstiegspunkt* bildet die mit *Platform/Application Integration* bezeichnete Ebene, die für die Bereitstellung und Aufbereitung kanalspezifischer Informationsinhalte und Integration der Applikationen zuständig ist. Die Applikationsschicht (*Application Layer*) beinhaltet die Oberflächensteuerung der Anwendungen für unterschiedliche fachliche und technische Kanäle. Auf Ebene des (*Business Orchestration Layer*) leben die Prozesse, die durch das Frontend orchestriert werden. Diese sind nach Anwendungsfall (z.B. „Tarifierung einer Lebensversicherung“) und nach Kanal (z.B. „Außendienst“) gegliedert. Es ist sinnvoll, auf dieser Ebene *Business Process Engines* zu verwenden, da diese zu einfacher anpassbaren Systemen führen. Hier werden Services des *Business Services Layer* genutzt, welche den Zugang zur eigentlichen Business-Logik und den Daten in den Backendsystemen regeln.

Diskussion: Essentiell bei diesem Modell ist, dass die Schichten voneinander entkoppelt sind und so eine Variabilität möglich ist. Eine übergeordnete Schicht kann durch eine alternative Lösung ersetzt oder um eine neue Variante ergänzt werden. Dadurch wird die Funktionalität der unteren Schicht wiederverwendet. Dies ist insbesondere für die Integration neuer Kanäle wichtig. Mit diesem Modell lassen sich die von den Schichten zu erfüllenden Aufgaben und verbundene Multikanalanforderungen, aber auch infrastrukturelle Fragen (Application Server, BPM System, CMS, etc.) diskutieren.

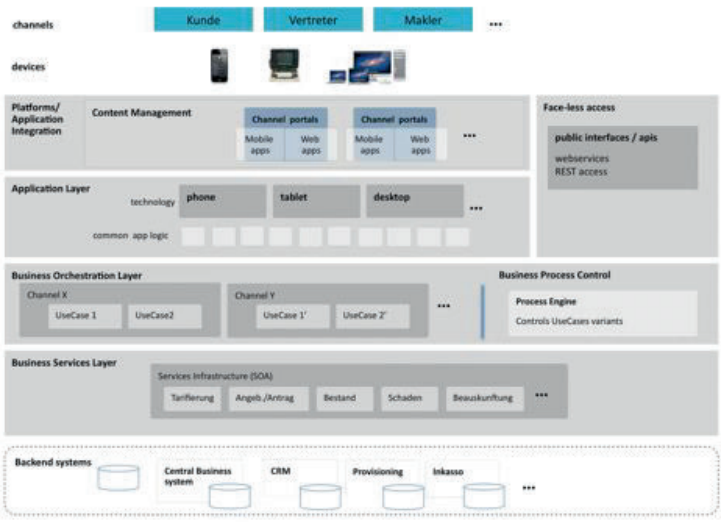


Abbildung 1: Basismodell für eine Multikanalarchitektur

3.2 Lösungsidee zur Variantenbildung

Das gefundene Basismodell legt eine nähere Untersuchung zur Variantenbildung insbesondere auf dem *Application Layer (AL)* und dem *Business Orchestration Layer (BOL)* nahe, da hier die (kanalspezifischen) Anpassungen häufiger vorkommen (s. auch [BGK+97]). Auf dem BOL kann eine Variantenbildung dadurch erreicht werden, dass Prozesse zu einem Anwendungsfall in *Basis-* und *kanalspezifischen* Anteil gegliedert werden. Der Basisprozess wird mit Erweiterungspunkten (EP) versehen, an denen kanalspezifische zusätzliche Prozesschritte eingehängt werden können. Für Oberflächen wird entgegen den modellierenden Ansätzen in Konzepten wie z.B. WebML ([CDM03]) eine automatisierte Erzeugung verfolgt (Ansätze hierzu in [BGL06]), welche mit den Eigenschaften der im Rahmen der Prozesse bearbeiteten Daten arbeitet. Wir gehen davon aus, dass auf der Obermenge der Daten aller Teilprozesse mit weiteren Eigenschaften versehen (Datenkohäsion, Kanalrelevanz,. . .) die Oberflächen kanalspezifisch erzeugt werden können.

3.3 Untersuchungen zu Oberflächen in multikanalfähigen Anwendungen

Das Ziel war es, einen ersten Schritt für die Oberflächen-Konzepte zu tun, der technologieneutrale, kanalspezifische Dialoge bereitstellt. Im ersten Schritt wird durch eine technologieneutrale, qualitative Beschreibung der Oberflächen die einfache Wiederverwendung in unterschiedlichen Kontexten gezeigt. Für die Machbarkeitsanalyse fand hier eine Be-

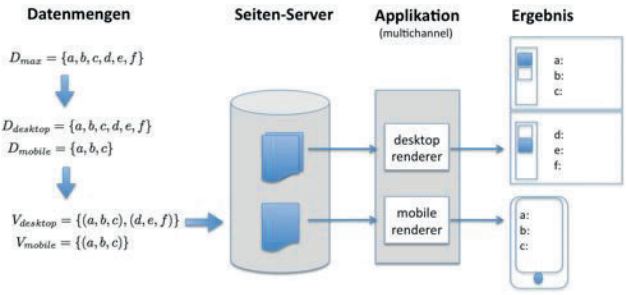


Abbildung 2: Oberflächen in einer Multikanalanwendung

schränkung auf die technischen Kanäle *mobile* und *desktop* statt.

Das Konzept basiert auf der Annahme, dass ein Prozess (hier Tarifierungsbaustein) über alle Kanäle hinweg auf einer Maximalmenge an Daten operiert (D_{max}). D_{max} ist bestimmt von der Summe der Eingabedaten, welche von den im Prozessverlauf verwendeten Backendservices benötigt werden. Unterschiedliche Kanäle arbeiten auf Untermengen dieser Menge ($D_{desktop}$, D_{mobile}), indem sie nur Teile dem Benutzer präsentieren (z.B. werden auf mobilen Geräten üblicherweise weniger Daten vom Benutzer eingefordert als bei Desktopbrowser-Oberflächen). Diese Untermengen werden zudem für Kanäle auf unterschiedlichen Dialogseiten/Sichten gruppenweise abgefragt bzw. dargestellt ($V_{desktop}$, V_{mobile}). Ergänzt um Typ- und allgemeine Darstellungsinformationen können diese Seitenbeschreibungen nun zum Aufbau der Dialoge in einer GUI-Technologie herangezogen und für das Endgerät gerendert werden (s. Abbildung 2).

Diskussion: Mit einem der gewählten UI-Technologie entsprechenden Renderer gestattet es der Ansatz, einfach Oberflächen für neue Kanäle zu definieren oder Bestehende für unterschiedliche technische Kanäle zu verwenden. Die Operationen, die ggf. mit Oberflächenelementen verknüpft sind (Buttons etc.) müssen damit umgehen können, dass nur Subsets der Daten zur Verfügung stehen oder kanalspezifische Implementierungen an die Oberflächen gebunden werden. Der Ansatz wurde bereits prototypisch umgesetzt und für client-seitige UI-Technologien (im Prototyp Sencha ExtJS/touch) verifiziert.

4 Ausblick

Die bisherigen Untersuchungen im GUI-Bereich sind vielversprechend. Hier muss eine weitere Verfeinerung hin zur Lösungsidee erfolgen und der bisherige Schritt der „Seitendefinition“ mittels Wissen über die Beziehungen der Daten untereinander etc. durch weitergehende Automatisierung ersetzt werden. Das Thema der Variantenbildung bei Prozessen ist zu untersuchen. Zudem müssen die Konzepte in die Gesamtarchitektur eingebettet werden.

Literatur

[Ban01] Frank Bannister. Dismantling the silos: extracting new value from IT investments in public administration. *Information Systems Journal*, 11:65–84, 2001.

[BGK+97] Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle und Heinz Züllighoven. Framework Development. *Communications of the ACM*, 40(10):52–59, 1997.

[BGL06] Matthias Book, Volker Gruhn und Matthias Lehmann. Automatic dialog mask generation for device-independent web applications. In *Proceedings of the 6th international conference on Web engineering - ICWE '06*, Seiten 209–216, New York, New York, USA, 2006. ACM Press.

[CDM03] Stefano Ceri, Florian Daniel und Maristella Matera. Extending WebML for Modeling Multi-Channel Context-Aware Web Applications. In *Fourth International Conference on Web Information Systems Engineering Workshops, 2003. Proceedings.*, Seiten 225–233, 2003.

[DR09] Peter Dadam und Manfred Reichert. The ADEPT project: a decade of research and development for robust and flexible process support. *Computer Science - Research and Development*, 23(2):81–97, April 2009.

[JrTD04] Ivar Jø rstad, Do Van Thanh und Schahram Dustdar. Towards Service Continuity for Generic Mobile Services. In *IFIP International Conference on Intelligence in Communication Systems (INTELLCOMM 04)*, 2004.

[KLKB07] Chang-ki Kim, Dan-hyung Lee, In-young Ko und Jongmoon Baik. A Lightweight Value-based Software Architecture Evaluation. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, Seiten 646–649, 2007.

[MP02] Guido Menkhaus und Wolfgang Pree. A Hybrid Approach to Adaptive User Interface Generation. In *Proceedings of the 24th International Conference on Information Technology Interfaces, 2002. ITI 2002.*, Seiten 185–190, 2002.

[Per06] Barbara (Ed.) Pernici. *Mobile Information Systems - Infrastructure and Design for Adaptivity and Flexibility*. Springer, 2006.

[WKNL07] Matthias Wieland, Oliver Kopp, Daniela Nicklas und Frank Leymann. Towards Context-aware Workflows. In *CAISE'07 Proceedings of the Workshop and Doctoral Consrtium*, Seiten 1–15, 2007.

[YN08] Takuto Yanagida und Hidetoshi Nonaka. Architecture for Migratory Adaptive User Interfaces. In *8th IEEE International Conference on Computer and Information Technology, 2008. CIT 2008.*, Seiten 450–455, 2008.

[ZDGH05] Olaf Zimmermann, Vadim Doubrovski, Jonas Grundler und Kerard Hogg. Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario : Rationale , Concepts , Lessons Learned. In *OOPSLA '05 Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 301–312, 2005.

Scribble - A Framework for Integrating Intelligent Input Methods into Graphical Diagram Editors

Andreas Scharf

andreas.scharf@cs.uni-kassel.de

Abstract: Creating modern software is a challenging but also a very creative task. Especially in early development phases like requirements engineering or architectural design software engineers use different mediums to manifest their thoughts and to discuss possible ambiguities. These mediums range from analog tools like pen & paper or whiteboards to digital ones like tablet pc's or smartboards. Whereas editing capabilities for analog mediums are restricted to add/remove operations, there already is great support in the digital world to later move, rotate or even share thoughts and diagrams with distributed teams. In addition the tool support for creating complex diagrams used to express software architecture and design along with sophisticated techniques like code generation is large. However, most of these tools restrict the user input to valid data, decreasing the software engineers flexibility which is why they often fall back to non formal tools. This doctoral thesis aims to combine the flexibility of informal sketching with the power of formal software engineering tools. As part of this thesis, a new generic framework will be created which dynamically augments new and already existing diagram editors with sketch-based input features.

1 Motivation

Building modern software is a complex but also a very creative task and a lot of frameworks for solving common problems in nearly every domain are available. On the one hand these frameworks simplify the development of big and more stable software but on the other hand one has to check thoroughly how to design and interconnect different software components. A good example are web frameworks: even though they support developers creating amazing desktop like applications, they also introduce great complexity like new communication concepts or connecting systems written in different programming languages. Therefore, writing the actual code is not the first task of software engineers in most cases. In fact, there are usually a couple of steps before the code writing phase like requirements engineering and architectural design. Especially during these early stages of development software engineers use different mediums to manifest their thoughts, discuss architecture and design with their team members or to clarify ambiguous interpretations of a conversation [MCK07]. Informal analog tools like pen & paper or whiteboards provide a great degree of freedom concerning allowed content but lack editing flexibility like copy/paste, scale and rotate or even share sketched content with distributed teams. With digital sketch-input enabled devices like tablet pc's or smartboards it is possible to overcome these problems [JINW04] and a remarkable amount of work has already been done

to further support this task [GSW01, MIEL99]. However, most approaches focus on creating and editing informal sketches while developers often use formal diagram editors for modifying diagram types like UML class diagrams or sequence diagrams. On the one hand these tools usually provide sophisticated editing-, validation- and code generation support but on the other hand they mostly stick to a strict syntax which limits the developers creativity by restricting editing operations to valid data. For that reason, developers fall back to more informal tools in many cases. Due to that, a lot of work has to been done twice since people first manifest their thoughts using pen & paper for instance and digitize the result afterwards to benefit from the mentioned tool support.

Therefore it is desirable to combine the strengths of software engineering tools with the flexibility of informal sketching. Several approaches addressing this problem have emerged, e.g. [QJJ03, PPY10] which provide support for recognizing hand drawn content. But most of the work either focuses on providing sketching capabilities for a single diagram tool or lack user control over several recognition aspects like used algorithms or how formalization happens. Marama [GHNN06] and SKETCH [SB10] both try to provide generic sketch support for graphical diagram editors in the Eclipse [1] environment. Marama is a diagram editor generation framework which supports developers in creating graphical editors. Sketch-input capabilities are restricted to editors generated by Marama. SKETCH on the other hand aims to add sketching features into new and already existing diagram editors based on the Eclipse Graphical Editing Framework (GEF). However, development of this framework is stuck and no information about extensibility and flexibility is given.

In the next section the resultant research questions are presented and the approach to answer these questions will be outlined.

2 Research questions and Approach

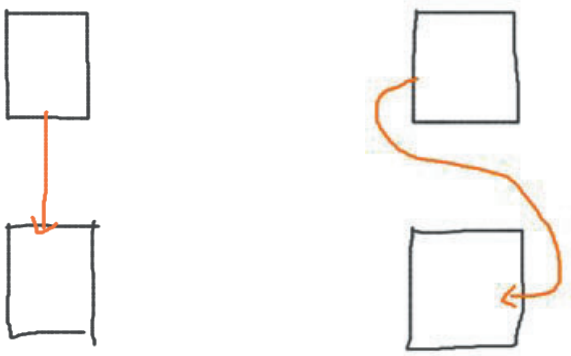
Although there already has been a remarkable amount of work in sketch-input related research areas, there are still a lot of open research questions in different categories. Some of them are of a very technical nature whereas others are more general. In the following the terms *sketch-based* and *scribble-based* are used synonymously and are related to hand drawn input whereas *Scribble* refers to the framework which should be created within the scope of this doctoral thesis.

- *Can complex graphical diagram editors be seamlessly augmented to support sophisticated sketch-based input? If yes, how and to what degree?*
- *How should a framework be designed to be useful for different graphical editor frameworks in different environments?* Are there any common components that can be used cross-platform? Example environments to evaluate include (but are not limited to) Eclipse and Visual Studio [2].

[1] `http://www.eclipse.org`
[2] `http://www.microsoft.com/visualstudio/eng`

- *What kind of extension points does the framework have to provide?* Which parts of the framework should be extensible, configurable or even exchangeable?
- *What are "intelligent" input methods?* How can different input capabilities of different devices like single-touch capable smartboards or multi-touch capable tablets be addressed? What about speech-input? Is it possible and reasonable to combine different kinds of input methods?
- Various questions concerning technical and usability aspects of the Scribble framework:
  - *Can incremental formalization be integrated?* At what exact points in time is recognition of sketched input performed?
  - *Can formalized and sketched input co-exist?* This is a technical question on the one hand but a usability question on the other hand. Do users want to be able to switch between these two forms of visualization?
  - *Is it possible to provide (auto-) correction mechanisms of unrecognized or wrong recognized elements?* It is possible that the meaning of a sketch is not clear at the point the user created it but elements created subsequently may give hints to what the initial sketch was meant to be.
  - *Is it possible to provide convenient text input mechanisms?* The core of this question is: How can text input be distinguished from all other sketched input? A related problem is the technical support for recognizing handwritten text. Even though there are plenty algorithms that claim to support text recognition, only a few perform well.
- *How can different types of edges be recognized?* Recognizing different types of nodes is well supported by a lot of algorithms even without a large amount of training data. However, recognizing edges is a different kind of recognition problem and cannot be solved with classical template matching based approaches. Figure 1 visualizes this problem.
- *How much training is needed to provide acceptable recognition results?* Who is training the algorithms? Can training data be shared?
- *Is scribble-based modeling accepted by modelers? What would be additional requirements on such a framework and its usability?*

To answer the research questions above, a requirement analysis has to be done first. This starts with asking software engineers about their opinion of usability aspects in the "end user" role on the one hand and what features they would like to have as integrators in the "developer" role on the other hand. These information would give some kind of feature list and also provide first hints of potential feature requirements of "client-frameworks" like Eclipse or Visual Studio. Also the extensible or exchangeable Scribble framework parts can be identified.



(a) Straight edge of type E between two nodes (b) Curved edge of type E between two nodes

Figure 1: The edge problem: two differently drawn edges of the same type E.

Analyzing different graphical editor creation frameworks along with diagram editors produced with these APIs would be the next step to get information about similarities and differences. All similarities are potential candidates for common components which could be used cross-platform. Furthermore, this analysis can be compared against the above mentioned requirement analysis to answer the question if it possible to support sophisticated sketch-based input within the evaluated client-frameworks and to what degree.

To support different input devices like single-touch tablet pc's or multi-touch capable smartboards along with a possible combination of speech-input, different use-cases covering common usage aspects have to be created. An example use-case would be "Move a node" which can be run on single- and multi-touch capable devices leading to different behavior of the Scribble framework. In this example, the user could use the common "two finger swipe" gesture to move a node on a multi-touch device. For single-touch devices some additional UI might be used to distinguish sketch-input from move commands.

Integrating sketch-, text- and speech-input recognition first involves some research of existing approaches and a study of their performance (qualitatively and quantitatively) and their eligibility to be integrated or extended. Also the research question how different types of edges could be recognized is part of this phase. Depending on the requirement analysis above, these parts are also subject to be extensible or exchangeable.

A very important question concerns the amount of needed training data and if this data can be shared. If it is reasonable to share training data amongst users, it would make sense to evaluate existing training data providers if any or create a new service for that purpose.

During the above mentioned research and different analysis, a prototype of the Scribble framework can be created to get a first picture to what extend all requirements can be covered. This prototype can be used to evaluate the different requirements of the end user and developer role. This evaluation also gives first results to the question if scribble-based modeling is accepted by modelers and might also reveal additional requirements.

3 Status Quo and Future Work

A first requirement analysis to create a feature list for both end user and developer role has already been done but this list is expected to change as mentioned in the last chapter. Related work like Marama and SKETCH have been evaluated and suitable algorithms for recognizing hand drawn content where identified.

The GEF framework in the Eclipse environment was the first candidate to analyze. A strategy to dynamically inject sketch-based input features into GEF along with a first prototype of the Scribble framework has been created. This work was submitted as a technical research paper [SAon] to the International Conference on Software Engineering 2013 and has been accepted for presentation in San Francisco from May 18th - 26th.

Evaluation of previous work in the domain of sharing training has revealed that it is reasonable to share such data in some kind of online database [FGP+09]. Although there is already some training data available, this data contains material only for primitive types like rectangles or ellipses in most cases. For that reason an "online training center" called WebScribble [Sei12] has been created as part of a bachelor thesis. WebScribble is a web application to create new diagram editor types along with supported node and edge types. For each node and edge type the user can add sample sketches and associate these sketches with the appropriate type. All results are persisted in a database whose content is planned to be accessible in form of a web service. This training data can then be imported into the Scribble framework and decreases the amount of required training material for other users.

The next steps include the evaluation of Visual Studio as another environment to create graphical diagram editors for. This will answer the question if common Scribble framework components are technically reasonable. A first look at available approaches for recognizing edges was made. This is a difficult problem since the same edge can look arbitrarily different and simple template matching algorithms cannot be used here. More work has to be investigated into this field and perhaps a new edge recognition algorithm has to be created. Also some currently available approaches for text recognition have been analyzed. The performance of most approaches is relatively worse and not usable in productive environments. Microsoft's text recognition capabilities on tablet pc's are excellent but only available in windows machines. However, the support of text-input methods is a crucial feature for some diagram types like UML class diagrams. Therefore more work has to be investigated here as well. A possible integration of speech-input is planned but no work has been done in this field yet.

The Scribble framework prototype is designed to work well with single-touch capable devices. If a device supports multi-touch, this feature should be utilized to increase user experience which also is future work.

At last, an evaluation of the final Scribble prototype has to be undertaken and suitable methods of measurement have to be found to answer the above mentioned research questions. It is planned to let students and professional developers test Scribble to see if the framework can be integrated easily into existing diagram editors and if requirements concerning usability are met.

References

[FGP+09] Martin Field, Sam Gordon, Eric Peterson, Raquel Robinson, Thomas Stahovich, and Christine Alvarado. The Effect of Task on Classification Accuracy: Using Gesture Recognition Techniques in Free-Sketch Recognition, 2009.

[GHNN06] J. Grundy, J. Hosking, Nianping Zhu, and Na Liu. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. In *21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE '06.*, pages 25–36, 2006.

[GSW01] François Guimbretière, Maureen Stone, and Terry Winograd. Fluid interaction with high-resolution wall-size displays. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, UIST '01, pages 21–30. ACM, 2001.

[JINW04] Wendy Ju, Arna Ionescu, Lawrence Neeley, and Terry Winograd. Where the wild things work: capturing shared physical design workspaces. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 533–541. ACM, 2004.

[MCK07] Gina Venolia Rob DeLine Mauro Cherubini and Andrew J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 557–566, 2007.

[MIEL99] Elizabeth D. Mynatt, Takeo Igarashi, W. Keith Edwards, and Anthony LaMarca. Flatland: new dimensions in office whiteboards. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, CHI '99, pages 346–353. ACM, 1999.

[PPY10] Beryl Plimmer, Helen C. Purchase, and Hong Yul Yang. SketchNode: intelligent sketching support and formal diagramming. In *Proceedings of the 22nd Conference of the Computer-Human Interaction Special Interest Group of Australia on Computer-Human Interaction*, OZCHI '10, pages 136–143. ACM, 2010.

[QJJ03] Qi Chen, John Grundy, and John Hosking. An E-whiteboard application to support early design-stage sketching of UML diagrams. In *In Proceedings of the 2003 IEEE Conference on Human-Centric Computing*, pages 219–226. IEEE CS Press, 2003.

[SAon] A. Scharf and T. Amma. Dynamic Injection of Sketching Features into GEF based Diagram Editors. Accepted at International Conference on Software Engineering 2013, submitted for publication.

[SB10] Ugo Braga Sangiorgi and Simone D.J Barbosa. SKETCH: Modeling Using Freehand Drawing in Eclipse Graphical Editors, 2010.

[Sei12] M. Seiler. WebScribble - Entwurf und Realisierung einer Webanwendung zur Erstellung von grafischen Editoren durch Freihandzeichnungen, 2012. thesis.

GI-Edition Lecture Notes in Informatics

P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.

P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.

P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Lan-guage to Information Systems, NLDB'2001.

P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensorgestützte Chirurgie; Workshop des SFB 414.

P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.

P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.

P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.

P-8 Reinhard Keil-Slawik, Johannes Magenheim (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.

P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.

P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.

P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.

P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.

P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.

P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.

P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.

P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Business 2002.

P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.

P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.

P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund.

P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3. Okt. 2002 in Dortmund (Ergänzungsband).

P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.

P-22 Sigrid Schubert, Johannes Magenheim, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.

P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit

P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen

P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003

P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web

P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin

P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Er-fahrungen und Visionen

P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems

P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications

P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometrics and Electronic Signatures

P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003

P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft

P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)

P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)

P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenberg (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik

P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik

P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003

P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003

P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004

P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing

P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste

P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistance, Scalability, Transactions – Database Mechanisms for Mobile Applications

P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning. E-Services

P-45 Bernhard Rumpe, Wofgang Hesse (Hrsg.): Modellierung 2004

P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment

P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society

P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications

P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven

P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik

P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004

P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration

P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Adhoc-Netze Medienintegration

P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government

P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen

P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality

P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Ranneberg, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments

P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für inforrmatische Bildung

P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen

P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit

P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and ist Applications

P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005

P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web

P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik

P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)

P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)

P-69 Robert Hirschfeld, Ryszard Kowalcyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005

P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)

P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005

P-72 Klaus P. Jantke, Klaus-Peter Fähnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment

P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): “Heute schon das Morgen sehen“

P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology

P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture

P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz

P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006

P-78 K.-O. Wenkel, P. Wagner, M. Morgenstern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel

P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006

P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce

P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS´06

P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006

P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics

P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications

P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems

P-86 Robert Krimmer (Ed.): Electronic Voting 2006

P-87 Max Mühlhäuser, Guido Rößling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik

P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006

P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur

P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006

P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006

P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1

P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2

P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen

P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies

P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices

P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik

P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik

P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten

P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments

P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)

P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme

P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007

P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops

P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications

P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures

P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1

P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2

P-111 Christian Eibl, Johannes Magenheim, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik

P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis

P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)

P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics

P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007

P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007

P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and manegement The 4th International Conference on Grid Service Engineering and Management GSEM 2007

P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007

P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications

P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges

P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik

P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)

P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Speciel Interest Groups PARS and PARVA

P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT:
Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung

P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)

P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008

P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany

P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems

P-130 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
1. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung

P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting. CC

P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik

P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 1

P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 2

P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik –
Aktuelle Forschungsergebnisse

P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008

P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures

P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008

P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschaftsinformatik

P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics

P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informations-systeme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management

P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung

P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)

P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management

P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)

P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009

P-148 Christian Erfurth, Gerald Eichler, Volkmar Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I^{2}CS 2009

P-149 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
2. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband

P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software

P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009

P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik

P-154 Stefan Fischer, Erik Maehle Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben

P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«

P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics 2009

P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture Reloaded – Informationsgestützte Landwirtschaft

P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010

P-160 Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner (Hrsg.)
Software Engineering 2010 – Workshopband
(inkl. Doktorandensymposium)

P-161 Gregor Engels, Dimitris Karagiannis Heinrich C. Mayr (Hrsg.)
Modellierung 2010

P-162 Maria A. Wimmer, Uwe Brinkhoff, Siegfried Kaiser, Dagmar Lück-Schneider, Erich Schweighofer, Andreas Wiebe (Hrsg.)
Vernetzte IT für einen effektiven Staat
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2010

P-163 Markus Bick, Stefan Eulgem, Elgar Fleisch, J. Felix Hampe, Birgitta König-Ries, Franz Lehner, Key Pousttchi, Kai Rannenberg (Hrsg.)
Mobile und Ubiquitäre Informationssysteme
Technologien, Anwendungen und Dienste zur Unterstützung von mobiler Kollaboration

P-164 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2010: Biometrics and Electronic Signatures Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-165 Gerald Eichler, Peter Kropf, Ulrike Lechner, Phayung Meesad, Herwig Unger (Eds.)
10th International Conference on Innovative Internet Community Systems (I^2CS) – Jubilee Edition 2010 –

P-166 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
3. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-167 Robert Krimmer, Rüdiger Grimm (Eds.)
4th International Conference on Electronic Voting 2010
co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-168 Ira Diethelm, Christina Dörge, Claudia Hildebrandt, Carsten Schulte (Hrsg.)
Didaktik der Informatik
Möglichkeiten empirischer Forschungsmethoden und Perspektiven der Fachdidaktik

P-169 Michael Kerres, Nadine Ojstersek Ulrik Schroeder, Ulrich Hoppe (Hrsg.)
DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V.

P-170 Felix C. Freiling (Hrsg.)
Sicherheit 2010
Sicherheit, Schutz und Zuverlässigkeit

P-171 Werner Esswein, Klaus Turowski, Martin Juhrisch (Hrsg.)
Modellierung betrieblicher Informationssysteme (MobIS 2010)
Modellgestütztes Management

P-172 Stefan Klink, Agnes Koschmider Marco Mevius, Andreas Oberweis (Hrsg.)
EMISA 2010
Einflussfaktoren auf die Entwicklung flexibler, integrierter Informationssysteme
Beiträge des Workshops der GI-Fachgruppe EMISA (Entwicklungsmethoden für Informationssysteme und deren Anwendung)

P-173 Dietmar Schomburg, Andreas Grote (Eds.)
German Conference on Bioinformatics 2010

P-174 Arslan Brömme, Torsten Eymann, Detlef Hühnlein, Heiko Roßnagel, Paul Schmücker (Hrsg.)
perspeGKtive 2010
Workshop „Innovative und sichere Informationstechnologie für das Gesundheitswesen von morgen“

P-175 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 1

P-176 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 2

P-177 Witold Abramowicz, Rainer Alt, Klaus-Peter Fähnrich, Bogdan Franczyk, Leszek A. Maciaszek (Eds.)
INFORMATIK 2010
Business Process and Service Science – Proceedings of ISSS and BPSC

P-178 Wolfram Pietsch, Benedikt Krams (Hrsg.)
Vom Projekt zum Produkt
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschafts-informatik (WI-MAW), Aachen, 2010

P-179 Stefan Gruner, Bernhard Rumpe (Eds.)
FM+AM`2010
Second International Workshop on Formal Methods and Agile Methods

P-180 Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, Holger Schwarz (Hrsg.)
Datenbanksysteme für Business, Technologie und Web (BTW)
14. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS)

P-181 Michael Clasen, Otto Schätzel, Brigitte Theuvsen (Hrsg.)
Qualität und Effizienz durch informationsgestützte Landwirtschaft, Fokus: Moderne Weinwirtschaft

P-182 Ronald Maier (Hrsg.)
6th Conference on Professional Knowledge Management
From Knowledge to Action

P-183 Ralf Reussner, Matthias Grund, Andreas Oberweis, Walter Tichy (Hrsg.)
Software Engineering 2011
Fachtagung des GI-Fachbereichs Softwaretechnik

P-184 Ralf Reussner, Alexander Pretschner, Stefan Jähnichen (Hrsg.)
Software Engineering 2011
Workshopband
(inkl. Doktorandensymposium)

P-185 Hagen Höpfner, Günther Specht, Thomas Ritz, Christian Bunse (Hrsg.)
MMS 2011: Mobile und ubiquitäre Informationssysteme Proceedings zur 6. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2011)

P-186 Gerald Eichler, Axel Küpper, Volkmar Schau, Hacène Fouchal, Herwig Unger (Eds.)
11th International Conference on Innovative Internet Community Systems (I^2CS)

P-187 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
4. DFN-Forum Kommunikations-technologien, Beiträge der Fachtagung 20. Juni bis 21. Juni 2011 Bonn

P-188 Holger Rohland, Andrea Kienle, Steffen Friedrich (Hrsg.)
DeLFI 2011 – Die 9. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. 5.–8. September 2011, Dresden

P-189 Thomas, Marco (Hrsg.)
Informatik in Bildung und Beruf
INFOS 2011
14. GI-Fachtagung Informatik und Schule

P-190 Markus Nüttgens, Oliver Thomas, Barbara Weber (Eds.)
Enterprise Modelling and Information Systems Architectures (EMISA 2011)

P-191 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2011
International Conference of the Biometrics Special Interest Group

P-192 Hans-Ulrich Heiß, Peter Pepper, Holger Schlingloff, Jörg Schneider (Hrsg.)
INFORMATIK 2011
Informatik schafft Communities

P-193 Wolfgang Lehner, Gunther Piller (Hrsg.)
IMDM 2011

P-194 M. Clasen, G. Fröhlich, H. Bernhardt, K. Hildebrand, B. Theuvsen (Hrsg.)
Informationstechnologie für eine nachhaltige Landbewirtschaftung
Fokus Forstwirtschaft

P-195 Neeraj Suri, Michael Waidner (Hrsg.)
Sicherheit 2012
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 6. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)

P-196 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2012
Proceedings of the 11th International Conference of the Biometrics Special Interest Group

P-197 Jörn von Lucke, Christian P. Geiger, Siegfried Kaiser, Erich Schweighofer, Maria A. Wimmer (Hrsg.)
Auf dem Weg zu einer offenen, smarten und vernetzten Verwaltungskultur
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2012

P-198 Stefan Jähnichen, Axel Küpper, Sahin Albayrak (Hrsg.)
Software Engineering 2012
Fachtagung des GI-Fachbereichs Softwaretechnik

P-199 Stefan Jähnichen, Bernhard Rumpe, Holger Schlingloff (Hrsg.)
Software Engineering 2012
Workshopband

P-200 Gero Mühl, Jan Richling, Andreas Herkersdorf (Hrsg.)
ARCS 2012 Workshops

P-201 Elmar J. Sinz Andy Schürr (Hrsg.)
Modellierung 2012

P-202 Andrea Back, Markus Bick, Martin Breunig, Key Pousttchi, Frédéric Thiesse (Hrsg.)
MMS 2012:Mobile und Ubiquitäre Informationssysteme

P-203 Paul Müller, Bernhard Neumair, Helmut Reiser, Gabi Dreo Rodosek (Hrsg.)
5. DFN-Forum Kommunikations-technologien
Beiträge der Fachtagung

P-204 Gerald Eichler, Leendert W. M. Wienhofen, Anders Kofod-Petersen, Herwig Unger (Eds.)
12th International Conference on Innovative Internet Community Systems (I2CS 2012)

P-205 Manuel J. Kripp, Melanie Volkamer, Rüdiger Grimm (Eds.)
5th International Conference on Electronic Voting 2012 (EVOTE2012)
Co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-206 Stefanie Rinderle-Ma, Mathias Weske (Hrsg.)
EMISA 2012
Der Mensch im Zentrum der Modellierung

P-207 Jörg Desel, Jörg M. Haake, Christian Spannagel (Hrsg.)
DeLFI 2012: Die 10. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V.
24.–26. September 2012

P-208 Ursula Goltz, Marcus Magnor, Hans-Jürgen Appelrath, Herbert Matthies, Wolf-Tilo Balke, Lars Wolf (Hrsg.)
INFORMATIK 2012

P-209 Hans Brandt-Pook, André Fleer, Thorsten Spitta, Malte Wattenberg (Hrsg.)
Nachhaltiges Software Management

P-210 Erhard Plödereder, Peter Dencker, Herbert Klenk, Hubert B. Keller, Silke Spitzer (Hrsg.)
Automotive – Safety & Security 2012
Sicherheit und Zuverlässigkeit für automobile Informationstechnik

P-211 M. Clasen, K. C. Kersebaum, A. Meyer-Aurich, B. Theuvsen (Hrsg.)
Massendatenmanagement in der Agrar- und Ernährungswirtschaft
Erhebung - Verarbeitung - Nutzung
Referate der 33. GIL-Jahrestagung
20. – 21. Februar 2013, Potsdam

P-213 Stefan Kowalewski, Bernhard Rumpe (Hrsg.)
Software Engineering 2013
Fachtagung des GI-Fachbereichs Softwaretechnik

P-215 Stefan Wagner, Horst Lichter (Hrsg.)
Software Engineering 2013
Workshopband
(inkl. Doktorandensymposium)
26. Februar – 1. März 2013, Aachen

The titles can be purchased at:

Köllen Druck + Verlag GmbH
Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
Fax: +49 (0)228/9898222
E-Mail: druckverlag@koellen.de

Gesellschaft für Informatik e.V. (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into
• seminars
• proceedings
• dissertations
• thematics
current topics are dealt with from the vantage point of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure high quality contributions.

The volumes are published in German or English.

Information: http://www.gi.de/service/publikationen/lni/

ISSN 1617-5468
ISBN 978-3-88579-609-1

The multi-conference "Software Engineering 2013" integrates the well-known scientific main conference, the industry-oriented "Software & Systems Engineering Essentials" and the education-oriented "Software Engineering im Unterricht der Hochschulen". This volume contains the proceedings of the colocated workshops and the doctoral symposium covering a wide range of topics including programming languages, process models, mobile systems, embedded systems, long-living systems, model-based development and traceability.