ISSN 1617-5468
ISBN 978-3-88579-607-7

This volume contains the contributions of the Software Engineering 2013 conference held in Aachen from February 26th to March 1st. The research, ideas, and experiences discussed at Software Engineering 2013 range from software quality and adaptive systems to model-based techniques and software product lines. The proceedings contain six invited contributions, fifteen refereed research papers, five refereed reports of research transfer to industry, and abstracts of the Software Engineering 2013 workshops.

Stefan Kowalewski, Bernhard Rumpe: Software Engineering 2013

213

GI-Edition

Lecture Notes
in Informatics

Stefan Kowalewski,
Bernhard Rumpe (Hrsg.)

Software Engineering 2013

Fachtagung des GI-Fachbereichs
Softwaretechnik

26. Februar – 1. März 2013
Aachen

Proceedings

Stefan Kowalewski, Bernhard Rumpe (Hrsg.)

Software Engineering 2013

Fachtagung des GI-Fachbereichs Softwaretechnik

26. Februar – 1. März 2013
in Aachen

Gesellschaft für Informatik e.V. (GI)

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

Volume P-213

ISBN 978-3-88579-607-7
ISSN 1617-5468

Volume Editors
Prof. Dr.-Ing. Stefan Kowalewski
Software für Eingebettete Systeme
Department of Computer Science 11
Ahornstr. 55, 52074 Aachen
Email: kowalewski@embedded.rwth-aachen.de
Prof. Dr. Bernhard Rumpe
Software Engineering
Department of Computer Science 3
RWTH Aachen University
Ahornstr. 55, 52074 Aachen
Email: rumpe@se-rwth.de

printed by Köllen Druck+Verlag GmbH, Bonn

Software – Innovator für Wirtschaft und Gesellschaft

Software ist mittlerweile nicht mehr nur Werkzeug und Hilfsmittel sondern treibende Kraft zahlreicher Innovationen in Wirtschaft und Gesellschaft. Innovationen in allen Lebensbereichen, beispielsweise durch ständig verfügbare Kommunikationsmöglichkeiten, effizientere und sichere Entwicklung von neuen Medikamenten, flexibleren Werkstoffen, energieeffizienteren Autos, autonom agierenden Robotern, intelligentem Energiemanagement, erdbebensicheren Bauwerken und smarten cyber-physischen Systemen treiben die Gesellschaft der Zukunft voran.

Als Objekt des Software Engineering ist Software auch Innovator für sich selbst geworden: Der aus zunehmender Softwarekomplexität wachsende Bedarf an Softwareentwicklern für verschiedene Anwendungsbereiche wird erst durch neue Ausbildungsmöglichkeiten und innovative Software Engineering Methoden ermöglicht. Die zunehmende Komplexität von Softwareprojekten in industriellen Anwendungen stellt hohe Anforderungen an Personen, Prozesse und Werkzeuge.

Die SE 2013 hat zum Ziel, neue wissenschaftliche Ergebnisse sowie industrielle Erfahrungen zu zukunftsweisenden Entwicklungsmethoden und Softwarelösungen aus verschiedenen Anwendungsbereichen zu diskutieren. Diese Diskussion dient der Förderung der Kommunikation zwischen Forschung und Anwendung, zwischen Hochschule und Industrie sowie dem einzelnen Softwareentwickler – eine notwendige Voraussetzung für Erhalt und Ausbau der Wettbewerbsfähigkeit unserer wirtschaftlichen aber auch unserer wissenschaftlichen Tätigkeiten.

Die Software Engineering-Tagungsreihe wird vom Fachbereich Softwaretechnik der Gesellschaft für Informatik e.V. getragen. 2013 haben die Lehrstühle Embedded Software und Software Engineering der RWTH Aachen die Gestaltung und Organisation der Tagung übernommen. Die SE 2013 bietet im Hauptprogramm drei spannende eingeladene wissenschaftliche Vorträge, vom Programmkomitee begutachtete detaillierte Berichte über laufende Forschungsarbeiten und -ergebnisse sowie Berichte über den Transfer von Forschungsergebnissen in die industrielle Praxis. Neu in das Programm aufgenommen wurden drei weitere eingeladene Vorträge über aktuelle laufende und frisch gestartete Großprojekte (ein Schwerpunktprogramm, ein SFB und ein Exzellenzcluster). Das Programmkomitee der SE 2013 hat insgesamt 20 wissenschaftliche Vorträge und Transferberichte ausgewählt. Flankiert wird die Konferenz durch neun Workshops und sechs Tutorials zu aktuellen, innovativen und praxisrelevanten Themen im Software Engineering. Abgerundet wird das Programm durch ein Doktorandensymposium, auf dem promovierende junge Wissenschaftlerinnen und Wissenschaftler ihre Arbeiten vorstellen und von erfahrenen Forscherinnen und Forschern konstruktive Rückkopplung zu ihren Dissertationsvorhaben erwarten, Schließlich wird zum ersten Mal ein Studierendentag abgehalten, an dem Studierende ihre Software Engineering Projekte vorstellen können.

Die Durchführung der Tagung Software Engineering 2013 wäre ohne die Mitwirkung der Sponsoren und vieler engagierter Personen nicht möglich gewesen. Wir bedanken uns daher bei allen Sponsoren, vor allem aber auch bei den vielen Helfern und Helferinnen der SE 2013:

- für die Organisation des Doktorandensymposiums Horst Lichter und Kurt Schneider,

- für die Koordination der Workshops und Tutorials Stefan Wagner,

- für die Organisation des Studierendentags Ulrik Schroeder,

- für die Mithilfe in den verschiedenen Programmkomitees,

Dank geht auch an die Unterstützung durch das Steering Committe, an den Fachbereich Softwaretechnik und die Organisatoren bei der Geschäftsstelle der GI.

Und letztlich ganz besonderer Dank an das Organisationskomitee, insbesondere Andreas Ganser, Thomas Gerlitz, Sylvia Gunder, Marina Herkenrath, Daniel Merschen, Anna Nawe, Jan Oliver Ringert, Stella Rahimi, Christoph Schulze, Andreas Wortmann und alle weiteren Helfer.

Ohne diese tolle Unterstützung wäre die SE 2013 nicht möglich.

Aachen, im Februar 2013

Stefan Kowalewski und Bernhard Rumpe
Tagungsleitung

Tagungsleitung
Stefan Kowalewski,
RWTH Aachen/Informatik 11 Embedded Software
Bernhard Rumpe,
RWTH Aachen/Informatik 3 Software Engineering

Leitung Workshops und Tutorials
Stefan Wagner,
Universität Stuttgart/Institut für Softwaretechnologie

Tagungsorganisation
Andreas Ganser,
Thomas Gerlitz,
Sylvia Gunder,
Marina Herkenrath,
Daniel Merschen,
Anna Nawe,
Jan Oliver Ringert,
Christoph Schulze,
Andreas Wortmann
(alle RWTH Aachen)

Programmkomitee
Colin Atkinson (Universität Mannheim)
Steffen Becker (Universität Paderborn)
Christian Berger (University of Gothenburg)
Manfred Broy (TU München)
Bernd Bruegge (TU München)
Jürgen Ebert (Universität Koblenz-Landau)
Gregor Engels (Universität Paderborn)
Holger Giese (HPI Universität Potsdam)
Martin Glinz (Universität Zürich)
Michael Goedicke (Universität Duisburg-Essen)
Radu Grosu (TU Wien)
Volker Gruhn (Universität Duisburg-Essen)
Wilhelm Hasselbring (Christian-Albrecht-Universität Kiel)
Maritta Heisel (Universität Duisburg-Essen)
Stefan Jähnichen (TU Berlin/FhG FIRST)
Matthias Jarke (RWTH Aachen)
Gerti Kappel (TU Wien)
Christian Kästner (Carnegie Mellon University)
Udo Kelter (Universität Siegen)
Jens Knoop (TU Wien)
Stefan Kowalewski (RWTH Aachen)
Claus Lewerentz (BTU Cottbus)
Horst Lichter (RWTH Aachen)

Peter Liggesmeyer (TU Kaiserslautern)
Florian Matthes (TU München)
Oscar Nierstrasz (Universität Bern)
Andreas Oberweis (KIT, Karlsruhe)
Barbara Paech (Universität Heidelberg)
Helmut Partsch (Universität Ulm)
Peter Pepper (TU Berlin)
Klaus Pohl (Universität Duisburg-Essen)
Alexander Pretschner (KIT, Karlsruhe)
Wolfgang Reisig (HU Berlin)
Ralf Reussner (KIT/FZI Karlsruhe)
Matthias Riebisch (Universität Hamburg)
Bernhard Rumpe (RWTH Aachen)
Gunter Saake (Universität Magdeburg)
Thomas Santen (European Microsoft Innovation Center)
Ina Schäfer (TU Braunschweig)
Wilhelm Schäfer (Universität Paderborn)
Bernhard Schätz (TU München)
Bastian Schlich (ABB Forschungszentrum Ladenburg)
Klaus Schmid (Universität Hildesheim)
Kurt Schneider (Leibniz Universität Hannover)
Andy Schürr (TU Darmstadt)
Friedrich Steimann (FU Hagen)
Gabriele Taentzer (Universität Marburg)
Stefan Tai (KIT/FZI Karlsruhe)
Birgit Vogel-Heuser (TU München)
Markus Voß (ACCSO)
Stefan Wagner (Universität Stuttgart)
Andreas Winter (Universität Oldenburg)
Mario Winter (FH Köln)
Uwe Zdun (Universität Wien)
Andreas Zeller (Universität des Saarlandes)
Heinz Züllighoven (Universität Hamburg)

Weitere Gutachter

Asim Abdulkhaleq
Alexander Bergmayr
Holger Breitling
Andrea Caracciolo
Marie Christin Platenius
Harald Cichos
Alexander Delater
Ana Dragomir
Kristian Duske
Zoya Durdik
Stephan Faßbender
Masud Fazal-Baqaie
Andreas Ganser
Tobias George
Christian Gerth
Christian Gierds
Mario Gleirscher
Regina Hebig
Joerg Henss
Stefan Hofer
Paul Huebner
Timo Kehrer
Andreas Koch
Anne Koziolek

Stephan Krusche
Philip Langer
Malte Lochau
Tanja Mayerhofer
Rene Meis
Benjamin Nagel
Stefan Neumann
Thomas Noack
Jan-Peter Ostberg
Birgit Penzenstadler
Pit Pietsch
Robert Prüfer
Jasmin Ramadani
Chris Rathfelder
Eugen Reiswich
Sascha Roth
Andreas Scharf
Alexander Schneider
Sandro Schulze
Norbert Siegmund
Jan Sürmeli
Thomas Thüm
Irina Todoran
Matthias Vianden
Andreas Vogelsang
Martin Wagner
Sebastian Wätzoldt
Thorsten Weyer
Gabriele Zorn-Pauli

Offizieller Veranstalter
Fachbereich Softwaretechnik der Gesellschaft für Informatik (GI)

Mitveranstalter
Rheinisch-Westfälische Technische Hochschule Aachen

PLATIN SPONSOREN

DSA

QSC AG

GOLD SPONSOREN

4SOFT Solid Innovation

ABB

adesso business. people. technology.

ANECON

arvato BERTELSMANN

Audi Electronics Venture GmbH

C1 WPS

CARMEQ

GENERALI Informatik Services

MathWorks

VOLKSWAGEN AKTIENGESELLSCHAFT

SILBER SPONSOREN

Accso Accelerated Solutions

IVU TRAFFIC TECHNOLOGIES AG

.msg systems

BRONZE SPONSOREN

iSQI International Software Quality Institute

dpunkt.verlag

IRIS Document to Knowledge PRODUCTS & TECHNOLOGIES

Inhaltsverzeichnis

SE | 13
SOFTWARE ENGINEERING

Eingeladene Vorträge

On-the-Fly Computing – Das Entwicklungs- und Betriebsparadigma für Softwaresysteme der Zukunft

Gregor Engels

Institut für Informatik
Universität Paderborn
Zukunftsmeile 1
33102 Paderborn
engels@upb.de

Extended Abstract

Alle Domänen und Branchen der heutigen Wirtschaft sind auf eine effiziente und effektive Entwicklung von benötigten Softwaresystemen angewiesen. Das 40 Jahre alte Prinzip der Beschaffung von Softwaresystemen durch den Einkauf von teuren, relativ unflexiblen Standardlösungen beziehungsweise der noch teureren Erstellung durch Softwarehäuser oder eigene Softwareabteilungen muss deshalb in Frage gestellt werden. Mit dem Einsatz von Cloud Computing-Techniken wird es möglich, Softwaresysteme und die für den Betrieb benötigten Ressourcen nur bei Bedarf und nur in der benötigten Form einzukaufen. Mit dem Ansatz der service-orientierten Architekturen stehen Methoden zur Verfügung, Software zumindest unternehmensintern flexibel zusammenzustellen.

Diese ersten Ansätze für eine neue Art der Entwicklung und des Betriebs von Softwaresystemen bilden den Ausgangspunkt für die Forschungen in dem seit 2011 laufenden DFG Sonderforschungsbereich (SFB) 901 „On-The-Fly Computing“ an der Universität Paderborn. Die Vision des On-The-Fly Computing ist, dass die Softwaresysteme der Zukunft aus individuell und automatisch konfigurierten und zur Ausführung gebrachten Softwarebausteinen bestehen, die auf Märkten frei gehandelt werden und flexibel kombinierbar sind.

Um zu erforschen, in wie weit diese Vision realisierbar ist, werden Konzepte, Methoden und Techniken entwickelt, die eine weitestgehend automatische Konfiguration, Ausführung und Adaption von Softwaresystemen aus auf weltweiten Märkten verfügbaren Services ermöglichen.

Um diese Ziele zu erreichen, arbeiten an der Universität Paderborn Informatiker aus unterschiedlichen Disziplinen wie Softwaretechnik, Algorithmik, Rechnernetze, Systementwurf, Sicherheit und Kryptographie mit Wirtschaftswissenschaftlern zusammen, die ihre spezifische Expertise einbringen, mit der die Organisation und Weiterentwicklung des Marktes vorangetrieben werden kann.

Der SFB strukturiert hierbei seine Arbeiten in drei Projektbereiche:

- Im Projektbereich A: Algorithmische und ökonomische Grundlagen für die Organisation großer, dynamischer Märkte werden neue Methoden des Distributed Computing erarbeitet, da die Größe und Dynamik dieser Netze von Akteuren und Services in den Märkten eine zentrale Steuerung unmöglich machen.
- Im Projektbereich B: Modellierung, Komposition und Qualitätsanalyse für das On-The-Fly Computing werden u.a. Ansätze der Softwaretechnik erarbeitet, die eine exakte Beschreibung, Analyse und Verifikation von funktionalen und nicht-funktionalen Aspekten von Services ermöglichen. Für die Konfiguration von komplexen Services werden neue Konzepte aus den Bereichen der Logik und heuristischen Suche erarbeitet.
- Im Projektbereich C: Verlässliche Ausführungsumgebungen und Anwendungsszenarien für das On-The-Fly Computing werden Fragen der Robustheit und (Angriffs-)Sicherheit von Märkten und Prozessen der Erbringung von Dienstleistungen sowie der Organisation hochgradig heterogener Rechenzentren, sog. On-The-Fly Compute Centers, untersucht. Des Weiteren ist ein Anwendungsprojekt integriert, das sich mit den Möglichkeiten zur Modellierung, Konfiguration und Ausführung von Optimierungssystemen für Versorgungs- und Logistiknetzwerke befasst.

Der Vortrag fokussiert auf die softwaretechnischen Herausforderungen des On-the-Fly Computing und die ersten im SFB erzielten Ergebnisse. Weitere Informationen zum SFB 901 findet man unter http://sfb901.uni-paderborn.de/sfb-901.

On Shared Understanding in Software Engineering

Martin Glinz[1], Samuel Fricker[2]

[1]Department of Informatics, University of Zurich, Switzerland
glinz@ifi.uzh.ch
[2]Software Engineering Research Laboratory, Blekinge Institute of Technology (BTH), Sweden
samuel.fricker@bth.se

Abstract: Shared understanding is essential for efficient communication in software development and evolution projects when the risk of unsatisfactory outcome and rework of project results shall be low. Today, however, shared understanding is used mostly in an unreflected, intuitive way. This is particularly true for implicit shared understanding.

In this paper, we investigate the role, value and usage of shared understanding in Software Engineering. We contribute a reflected analysis of the problem, in particular of how to rely on implicit shared understanding. We discuss enablers and obstacles, compile existing practices for dealing with shared understanding, and present a roadmap for improving knowledge and practice in this area.

1. Motivation

Shared understanding between stakeholders and software engineers is a crucial prerequisite for successful development and deployment of any software system. A *stakeholder* is a person or organization that has a (direct or indirect) influence on a system's requirements [GlW07]. *Software engineers* are the persons involved in the specification, design, construction, deployment, and maintenance/evolution of software systems. In traditional development projects, eliciting requirements and producing a comprehensive requirements specification serves for establishing shared understanding both among and between stakeholders (end users, customers, operators, managers, ...) and software engineers (requirements engineers, architects, developers, coders, testers,...). In agile environments, stories, up-front test cases, and rapid validation serve the same purpose.

Shared understanding among a group of people has two facets: *explicit shared understanding (ESU)* is about interpreting explicit specifications[1] (requirements, design documents, manuals,...) in the same way by all group members. On the other hand, *implicit shared understanding (ISU)* denotes the common understanding of non-specified facts, assumptions, and values. The shared context provided by implicit shared understanding

[1]In the normal case, explicit specifications are captured in writing. Principally, however, explicit verbal communication remembered by all team members is also a form of explicit shared understanding.

reduces the need for explicit communication [St12] and, at the same time, lowers the risk of misunderstandings.

In daily software engineering life, we make use of shared understanding without much reflection about it. *Explicit shared understanding* based on specifications is rather well understood today. In particular, research and practice in Requirements Engineering have contributed practices for eliciting requirements, documenting them in specifications and validating these specifications. In contrast, the role and value of *implicit shared understanding* is frequently neither clear nor reflected.

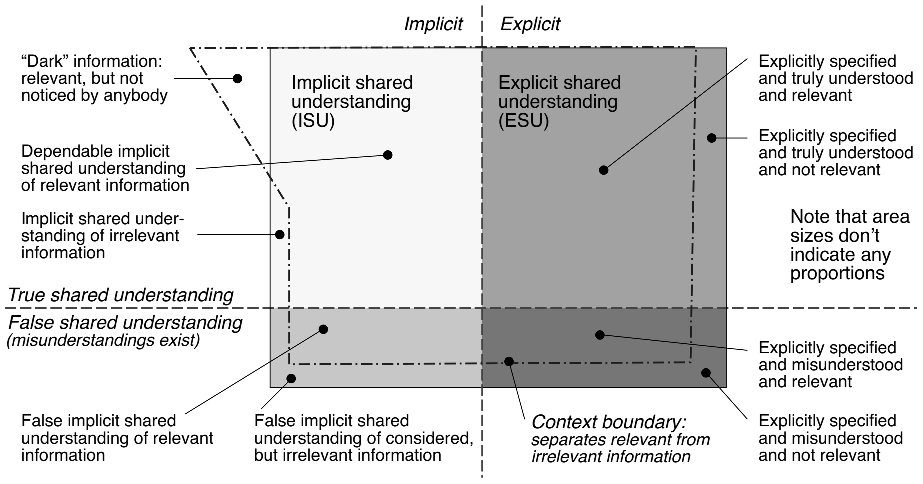
This paper contributes an essay on shared understanding in Software Engineering. We reflect about the role and value of shared understanding, identify enablers and obstacles for achieving shared understanding, and compile a list of practices related to shared understanding. We then focus on implicit shared understanding, reflecting about its value and risk and describing when and how we can (or even should) rely on implicit shared understanding.

Our work on shared understanding is rooted in the first author's previous work on alternative forms for specifying quality requirements which includes considerations about the cost and benefit of requirements [Gl08], and the second author's work on communicating requirements by handshaking with implementation proposals instead of writing large explicit specifications [FGB10][FG10].

The remainder of this paper reflects the role and value of shared understanding (Section 2), presents enablers, obstacles and a compilation of practices (Sections 3, 4), and then discusses how to rely on implicit shared understanding (Section 5). We conclude with a short look at explicit shared understanding (Section 6), a roadmap (Section 7), a brief look at related work (Section 8) and some concluding remarks.

2. The role and value of shared understanding

Shared understanding in Software Engineering always implies dealing with both explicit and implicit shared understanding. Figure 1 illustrates the various facets of shared understanding. It is important to note that shared understanding can be false. This means that the involved people believe to have a shared understanding of some items, while there are differences in the actual understanding. We illustrate this briefly, taking the problem of a road construction site with one-lane traffic controlled by two traffic lights as an example. Assume that we need to develop a traffic light control system for managing alternating one-lane traffic, with a team of stakeholders and developers. (1) If the notion that a red light is a stop signal is shared by all team members, we have *implicit shared understanding*. (2) If there exists an explicit requirement stating "The stop signal shall be represented by a red light", we have *explicit shared understanding*, provided that all team members interpret this requirement in the same way. (3) If nothing is specified about the go signal, because the stakeholders take it for granted that the go signal can be either a green light or a flashing yellow light, while the developers believe that the go signal must be implemented as a green light, we have a misunderstanding. If this misunderstanding goes undetected, we have *false implicit shared understanding*. (4) Assume there is an explicit requirement "The system shall support flashing yellow lights" without any further requirements about flashing yellow lights. In this case, a stakeholder might mean 'flashing yellow on one side and red on the opposite side' while a developer might interpret this as 'flashing yellow on both sides'. If this misunderstanding goes undetected, it results in *false explicit shared understanding*. Note that the area sizes in Fig. 1 don't indicate any proportions. We are not aware of any research investigating the percentages of information in the categories identified in Fig. 1.



Implicit
Explicit
"Dark" information: relevant, but not noticed by anybody
Implicit shared understanding (ISU)
Explicit shared understanding (ESU)
Explicitly specified and truly understood and relevant
Explicitly specified and truly understood and not relevant
Dependable implicit shared understanding of relevant information
Implicit shared understanding of irrelevant information
Note that area sizes don't indicate any proportions
True shared understanding
False shared understanding (misunderstandings exist)
Explicitly specified and misunderstood and relevant
False implicit shared understanding of relevant information
False implicit shared understanding of considered, but irrelevant information
Context boundary: separates relevant from irrelevant information
Explicitly specified and misunderstood and not relevant

Figure 1: Forms and categories of shared understanding

When developing a system, there is a *context boundary* that separates the information which is *relevant* for the system to be built from the irrelevant rest of the world (cf. Fig. 1). However, sometimes stakeholders or software engineers also consider information (and might even achieve shared understanding about it) that is actually irrelevant. Conversely, we may have *"dark" information* that would be relevant, but has gone unnoticed by all team members. For example, imagine that in the traffic light problem presented above, there is a legal constraint in some country that forbids the configuration 'flashing yellow on one side and red on the opposite side'. If nobody in the team is aware of this fact, this constraint is "dark" information. In this paper, we will not further elaborate the issue of relevant vs. irrelevant information and concentrate on the problem of shared understanding, regardless whether or not the underlying information actually is relevant.

Relying only on implicit shared understanding does not work because real-world software is too complex for being developed without any explicit documentation. Conversely, relying solely on explicit shared understanding is both impossible and economically unreasonable for any real world software system. It is *impossible* because even within the context boundary of a system, the amount of relevant information is potentially infinite. Even if we assume that a system can be specified completely within finite time and space bounds, such a complete specification wouldn‘t be *economically reasonable* in most cases: the cost of creating and reading a complete specification would exceed its benefit, i.e., making sure that the deployed system meets the expectations and needs of its stakeholders.

Relying on implicit shared understanding has a strong economic impact on software development: The higher the extent of implicit shared understanding, the less resources have to be spent for explicit specifications of requirements and design, thus saving both cost and development time. However, these benefits come with a serious threat: assumptions about the existence or the degree of implicit shared understanding might be false. In this situation, omitting specifications yields systems that don’t satisfy their stakeholders’ needs, thus resulting in development failures or major rework for fault fixing. There is another important caveat: even if, in a given project, we manage to rely on implicit shared understanding to a major extent, reflection and explicit documentation of key concepts such as system goals, critical requirements and key architectural decisions remain necessary. Otherwise, development team fluctuation as well as evolving the deployed system by people other than the original developers can easily become a hidden knowledge nightmare. In some situations it might even be useful to document which requirements and design decisions have not been documented in detail due to reliance on implicit shared understanding.

In summary, the problem of how to deal with shared understanding for ensuring successful software development can be framed as follows:

(P1) Achieving shared understanding by explicit specifications as far as needed,

(P2) Relying on implicit shared understanding of relevant information as far as possible,

(P3) Determining the optimal amount of explicit specifications, i.e., striking a proper balance between the cost and benefit of explicit specifications.

Note that P1, P2, and P3 are not orthogonal problems, but different views of the same underlying problem: *How can we achieve specifications that create optimal value? Value* in this context means the *benefit* of an explicit specification (in terms of bringing down the probability for developing a system that doesn’t satisfy its stakeholders’ expectations and needs to a level that one is willing to accept), and the *cost* of writing, reading and maintaining this specification.

P2 can be sub-divided into three sub-problems:

(P2a) Increasing the extent of implicit shared understanding,

(P2b) Reducing the probability for false assumptions about implicit shared understanding,

(P2c) Reducing the impact of (partially or fully) false assumptions about implicit shared understanding.

Again, these sub-problems are not orthogonal: for increasing the extent of implicit shared understanding (P2a), we need to control the risk of false shared understanding, which can be framed in terms of probability (P2b) and impact (P2c).

For addressing these problems, it is important to know about the enablers and obstacles for shared understanding (Sect. 3) and appropriate practices for dealing with shared understanding (Sect. 4).

3. Enablers and obstacles

This section provides a list of enablers and obstacles for shared understanding. Knowing about enablers and obstacles helps analyze a given project context with respect to the ease or the difficulty of relying on shared understanding. In a constructive sense, it helps setting up a software development project such that relying on shared understanding becomes easier and less risky.

Domain knowledge. Knowledge about the domain of the system to be built enables software engineers to understand the stakeholders' needs better, thus fostering shared understanding. Domain knowledge reduces the probability that software engineers misinterpret specifications or fill gaps in the specification in an unintended way.

With respect to implicit shared understanding, domain knowledge can also be a threat: for example, implicit domain assumptions may be taken for granted by some team members, although not everybody involved is aware of them. In this situation, a smart person without domain knowledge (a "smart ignoramus" as Berry calls it [Be02]) can be valuable. Having a "smart ignoramus" in the team actually is an enabler for shared understanding. By asking all those questions that domain experts don't ask because the answer seems to be obvious to them, misunderstandings about domain concepts are uncovered, thus improving shared understanding.

Previous joint work or collaboration. If a team of software engineers and stakeholders has collaborated successfully in previous projects, the team shares a lot of implicit understanding of the individual team members' values, habits, and preferences. In this situation, a rather coarse and high-level specification may suffice as a basis for successfully developing a system.

Existence of reference systems. When a system to be developed is similar to an existing system that the involved stakeholders and engineers are familiar with, this existing system can be used as a *reference system* for the system to be built. Such a reference system constitutes a large body of implicit shared understanding.

Culture and Values. When the members of a team are rooted in the same (or in a similar) culture and share basic values, habits, and beliefs, building shared understanding about a problem is much easier than it is for people coming from different cultures with different value systems. With increasing cultural distance between team members, the probability for missing or false implicit shared understanding is rising. The risk for misunderstanding explicit specifications is also higher than normal.

Geographic distance. Geographically co-located teams communicate and collaborate differently than teams where members live in different places and time zones. Geographic co-location reduces the cultural distance mentioned above, thus enabling and fostering shared understanding.

Trust. Mutual trust is a prerequisite for relying on implicit shared understanding. When involved parties, in particular customer and supplier, don't trust each other, explicit and detailed specifications for the system and the project must be created in writing, because everything that is not specified explicitly may not happen in the project, regardless of actual importance and needs.

Contractual situation. When the relationship between customer and supplier is governed by a fixed-price contract with explicitly specified deliverables, shared understanding must be established on the basis of explicit specifications; there is not much room left for implicit shared understanding. However, even when a project is fully governed by an explicit contract, some basic implicit understanding, particularly about meanings of terms as well as cultural, political and legal issues, must exist among the involved parties. For example, if a contractual requirements specification states requirements about an order entry form, the specification will typically not state that, for entering alphanumeric data into a field, the system has to position the cursor at the left edge of the field and display the data being typed from left to right. Instead, this is treated as a shared assumption about form editing.

Outsourcing. When significant parts of a system development are outsourced, there is a high probability of non-matching cultural backgrounds (in terms of values, habits, beliefs) among team members. Team members at remote places who are assigned to outsourced work packages may also lack domain knowledge. So outsourcing is a significant obstacle to shared understanding.

Regulatory constraints. If a system requires approval by a regulator, the regulator will typically require detailed, explicit specifications, thus leaving no room for alternative forms such as implicit shared understanding. So regulatory constraints are an obstacle to implicit shared understanding.

Normal vs. radical design. When the development of a system is governed by the principle of "normal design" [Vi93], i.e., both the problem and the solution stay within an envelope of well-understood problems and solutions, the degree of implicit shared understanding is typically much higher than in "radical design", where the problem, the solution or both are new. Conversely, radical design entails a higher probability for false shared understanding.

Team size and diversity. The larger and the more diverse a team, the more difficult it becomes to establish and rely on shared understanding. Hence, small teams are not only advantageous with respect to communication overhead, but also with respect to the ease of shared understanding.

Fluctuation. Fluctuation of personnel is another common obstacle. This is especially problematic for implicit shared understanding, independent of whether stakeholders or development team members change.

4. Practices for enabling, building, and assessing shared understanding

In this section, we compile a set of practices for dealing with shared understanding. We group them into three categories (Tables 1-3): *Enabling practices* lay foundations for shared understanding or are generic procedures for achieving or analyzing shared understanding. *Building practices* are directed towards achieving shared understanding, (i) by creating explicit artifacts, or (ii) by building a dependable body of implicit shared understanding. *Assessment practices* aim at determining to which extent the understanding of some artifact or topic is actually shared among a group of people involved. Some practices can be used both for building and assessing shared understanding.

Almost all practices listed in Tables 1-3 are well-known practices in Software Engineering that don't need further explanation. The added value of Tables 1-3 lies in the classification and characterization of the listed practices with respect to shared understanding. Potential usage of the practices will be discussed in Sect. 5.

Table 1: Enabling Practices

Practice	**Goal**	**For**[1]	**Based on**
Domain scoping	Identify/narrow the domain where shared understanding has to be achieved	ESU ISU	Discussion, Documents, Models
Stakeholder/project team member selection	Identify the stakeholders and project team members who will need to achieve shared understanding	ESU ISU	Stakeholder analysis, searching, team building
Domain understanding	Achieve general understanding of important domain concepts	ESU ISU	Discussion, Documents, Models
Collaborative learning	Increase the degree of ISU by a shared discourse of learned items	ISU	Moderated or free discourse about learned subjects
Feedback[2]	Ensure shared understanding between sender(s) and recipients(s) of information	ESU ISU	Communication, Artifacts
Team building	Build teams with shared experience and cultural background	ISU	Project management processes
Negotiation and prioritization	Achieve explicit consensus on some concept or issue	ESU	Artifacts and processes

[1]The form of shared understanding that the practice is useful for. ESU and ISU denote explicit shared understanding and implicit shared understanding, respectively.

[2]Feedback addresses both ESU and ISU, depending on what form of understanding communication is based on. It plays a key role in many of the building and assessment practices given in Tables 2 and 3.

Table 2: Building Practices

Practice	**Goal**	**For**	**Based on**
Domain modeling[1]	Achieve explicit shared understanding of important domain concepts	ESU ISU	Models, Documents
Problem/ solution modeling[1]	Achieve explicit shared understanding of system requirements or architecture	ESU ISU	Models, Documents
Holding workshops	Achieve consensus about an artifact (vision, requirements spec, architecture) among the persons involved	ESU ISU	Mainly discussion, also models, documents and examples
Building and using a glossary[1]	Achieve explicit shared understanding of the relevant terminology when developing a system	ESU ISU	Glossary document
Using ontologies	Achieve a general understanding of the major terms and concepts in a given domain	ISU	Documents containing the used ontologies
Formalizing requirements or architecture[1]	Achieve explicit shared understanding of requirements and/or architectural design	ESU ISU	Requirements specifications, system architecture
Quantifying requirements	Achieve explicit shared understanding of a quality requirement by quantifying it	ESU	Quality requirements
Prototyping[2]	Build shared understanding of requirements or designs by experiencing how the final system will look and work	ESU ISU	Prototype
Reference systems	Achieve shared understanding of a system by referring to an existing system that the involved persons are familiar with	ISU	Existing reference system
Handshaking [FGB10][3]	Achieve shared understanding in a software product management context by feeding goals to the developers by the product manager and feeding back implementation proposals	ISU ESU	Goals and implementation proposals

[1]Building explicit shared understanding by modeling, glossaries, requirements formalization, etc. also improves implicit shared understanding of non-specified or coarsely specified concepts.

[2]An approved prototype is an artifact that explicitly represents shared understanding of how a system to be shall look. On the other hand, a prototype also tests and fosters implicit shared understanding.

[3]Handshaking fosters implicit shared understanding. On the other hand, implementation proposals document the shared understanding explicitly.

Table 3: Assessment Practices

Practice	**Goal**	**For**	**Based on**
Creating and playing scenarios[1]	Assess and foster shared understanding about how a system will work in typical situations	ESU ISU	Scenarios, i.e., examples of system usage
Creating and (mentally) executing test cases[1]	Assess and foster shared understanding of the results that a system will produce in typical situations	ESU ISU	Test cases, i.e., examples of system usage
Model checking	Formally determine whether some understanding of a specification actually holds	ESU	Formal requirements
Checking the glossary	A good glossary lowers the probability of false shared understanding	ESU ISU	Glossary document
Prototyping or simulating systems[1]	Assess and foster shared understanding about how a system will work in typical situations	ESU ISU	Formal or semi-formal requirements
Short feedback cycles[2]	Minimize the timespan between making/ causing and detecting misunderstandings or errors	ESU ISU	Processes that enable and encourage rapid feedback
Paraphrasing[3]	Assess whether the understanding of the person(s) paraphrasing an artifact matches the understanding of the author(s) of the artifact	ESU ISU	Human-readable artifacts
Having a smart ignoramus in the team [Be02]	Uncover misunderstandings by asking all those questions that domain experts don't ask because the answers seems to be obvious to them	ESU ISU	Asking questions
Comparing to reference systems	Assess shared understanding of a system by comparing it to an existing system that the involved persons are familiar with	ISU	Existing reference system
Measuring shared understanding	Measure the degree of shared understanding for a given artifact or set of implicit concepts	ESU ISU	An artifact such as a requirements specification or implicit concepts
Measuring ambiguity [GW89]	Assess the ambiguity of requirements with polling (see Chapter 19 in [GW89]), thus indirectly assessing shared understanding	ESU ISU	Requirements, polling questions

[1]Addresses ESU by checking whether a group of involved people is understanding an explicit specification in the same way. Addresses ISU when there is no or only a coarse specification.

[2]Short feedback cycles aim at detecting misunderstandings rapidly, as well as keeping the impact of false assumptions low when relying on ISU.

[3]Mainly addresses ESU. Also useful for assessing ISU when communicating concepts orally.

5. Relying on implicit shared understanding

In this section we primarily address problem P2 posed in Section 2:

(P2) Relying on implicit shared understanding of relevant information as far as possible

As mentioned in Section 2, P2 can be divided into the sub-problems of increasing the degree of implicit shared understanding and controlling the risk of implicit shared understanding, i.e., reducing both the probability for and the impact of false assumptions about shared understanding.

5.1 Reducing the probability of false implicit shared understanding

5.1.1 Enabling practices

The enabling practices which address implicit shared understanding (cf. Table 1) contribute to the creation of a stable and dependable basis for implicit shared understanding. *Domain scoping* and *domain understanding* narrow the amount of domain knowledge to be shared and lay the foundation for successfully communicating domain concepts. *Stakeholder selection* identifies the stakeholder roles that matter for a system to be built and helps identify proper representatives for these roles. *Team building* aims at selecting and forming teams such that members have shared experience, cultural background, and values. *Collaborative learning* helps create a common background when it is not possible to select people who already have this common background. *Feedback* is a general enabler for building and checking shared understanding.

5.1.2 Building practices

The building practices (cf. Table 2) help create and improve implicit shared understanding, thus constructively lowering the probability of undetected misunderstandings.

Modeling (of domains, problems or solutions) makes the modeled concepts explicit and thus converts implicit shared understanding into explicit shared understanding. However, due to feasibility and economical reasons, models are almost never complete and frequently not detailed and/or formal enough for making everything explicit. Such models help infer and properly interpret non-modeled or only coarsely modeled concepts and increase the probability of interpreting them correctly, thus contributing to the creation of proper implicit shared understanding.

Modeling is a particular way of *formalizing requirements or architecture*. For any other form of formalization, the same arguments as for modeling apply with respect to implicit shared understanding.

Workshops, although primarily aiming at the creation of explicit shared understanding by creating artifacts, also foster implicit shared understanding and reduce the probability of misunderstandings as a by-product. Firstly, this is due to the same effect as described for modeling above. Secondly, well-moderated workshops implicitly contribute to the creation of a shared notion of goals, basic concepts, and values for the system to be built.

Glossaries and *ontologies* provide explicit definitions of terminology for the system to be built and its domain. As this constitutes again a conversion of implicit shared understanding into explicit shared understanding, the same arguments as given for models apply: explicitly shared terminology reduces the probability of misunderstandings when concepts using this terminology are not specified or only coarsely specified.

Prototypes implement a selected subset of a system to be built. While a prototype secures explicit shared understanding of all features implemented in the prototype, it also improves implicit shared understanding of non-implemented features if the system to be built is implemented in the spirit and general directions given by the prototype.

Reference systems can serve as an anchor point for implicit shared understanding. If all persons involved are familiar with the reference system, implicit shared understanding of concepts about the system to be built can be achieved by referring to comparable or similar concepts in the reference system. Note that working with reference system can also be used as an assessment practice (see below).

With *handshaking* [FGB10], implementation proposals make the development team's interpretation of requirements explicit. When used early in the development process, the feedback provided by the implementation proposals allows building shared understanding among the stakeholders and the development team about the stakeholders' intentions.

5.1.3 Assessment practices

All assessment practices aiming at implicit shared understanding (cf. Table 3) contribute to the detection of misunderstandings, thus analytically lowering the probability of false implicit shared understanding.

Creating and playing scenarios make stakeholder intentions tangible and comprehensible by working with concrete examples. If implicit shared understanding of some concept or component can be exemplified by a representative set of scenarios, misunderstandings will be detected. Thus, the probability of false implicit shared understanding can be lowered systematically and significantly.

Creating and (mentally) executing test cases on requirements specifications or system architectures also exemplify how a system should behave in a given situation. Again, if implicit shared understanding of some concept or component can be exemplified by a representative set of test cases, misunderstandings will be detected, thus lowering the probability of false implicit shared understanding.

Prototyping or simulating systems has similar effects as playing scenarios: both make intentions tangible and comprehensible by example. Thus, the arguments given above for scenarios apply.

Short feedback cycles enable rapid detection of problems, including false implicit shared understanding. When misunderstandings are detected and corrected rapidly, the probability of undetected misunderstandings is reduced.

While *paraphrasing* is primarily a practice for assessing shared understanding of documents (i.e., explicit shared understanding), it can also be harnessed for assessing implicit shared understanding. For example, a stakeholder tells a requirements engineer that s/he needs feature X. In order to detect potential misunderstandings about what X actually is, the stakeholder tells a short story characterizing X, the requirements engineer paraphrases this story in her or his own words and then the stakeholder checks the paraphrased story against her or his original intentions.

Comparing to a reference system also is a form of assessment by example. If all persons involved are familiar with the reference system, implicit shared understanding of concepts about the system to be built can be checked by comparing these concepts to corresponding concepts in the reference system. Thus misunderstandings of such concepts will become obvious and can be fixed.

Measuring shared understanding is the only practice which is not well developed and understood today. In [FG10] we have described two approaches towards measuring requirements understanding: (a) *Ability to execute*, where architects estimate their confidence for developing an accepted product, (b) *R-Cov*, the coverage of requirements with explicit design.

If requirements are ambiguous, there is a high probability for misunderstanding them. Thus, *measuring ambiguity* with polling [GW89] indirectly assesses shared understanding. If the ambiguity of an implicit requirement or a vaguely stated requirement is measured, implicit shared understanding is assessed with respect to this requirement.

The scenario and test practices are challenged with respect to assessing implicit shared understanding of *non-functional* concepts such as quality requirements or constraints, because these concepts are difficult to express in scenarios or to capture in test cases. In contrast, comparison to reference systems works well also for non-functional concepts. Prototyping and simulation take a middle ground with respect to assessment of non-functional concepts.

5.2 Reducing the impact of false implicit shared understanding

The impact of false implicit shared understanding is defined as the cost for detecting and correcting the underlying misunderstandings plus the cost incurred by (i) stakeholder dissatisfaction and (ii) re-doing the work which has become invalid due to the misunderstandings.

The practice of short feedback cycles (cf. Table 3) strongly influences impact by reducing the timespan between causing and detecting misunderstandings: the less time a misunderstanding has to unfold, the lower its impact.

Applying the other practices for assessing implicit shared understanding such as using scenarios and test cases or comparing to reference systems (cf. Table 3) as *early* as possible also contributes to impact reduction, as early detecting and fixing problems costs considerably less than when the same problems are detected only late in the development cycle.

There are software development practices, for example, refactoring or design for change, that lower the cost of rework when errors are detected. These practices also lower the impact of false shared understanding.

The most effective way, however, of reducing the impact of false implicit shared understanding is to base the specification of high-risk concepts on *explicit* shared understanding rather than on implicit shared understanding. Risk in this context means the risk that the system to be built does not satisfy its stakeholders' expectations and needs when it is eventually deployed. In [Gl08] we discuss techniques for risk assessment of requirements and factors influencing the risk. By confining the reliance on implicit shared understanding to concepts with low or medium risk, both the average and the worst case impact of false implicit shared understanding are also confined.

5.3 Processes supporting implicit shared understanding

Traditional software development processes strongly rely on explicit specifications, thus confining implicit shared understanding mostly to basic understanding of domain concepts and the interpretation of accidentally underspecified items.

Agile software development processes, on the other hand, strongly rely on implicit shared understanding, but without much reflection. Stories and system metaphors provide general directions. Shared understanding of the unspecified details is secured by writing up-front test cases and short feedback cycles. Other practices, for example comparison to reference systems, are not systematically used in agile development.

Any form of *incremental* or *prototype-oriented* development process has potential for relying on implicit shared understanding to a significant extent. However, contemporary process descriptions don't reflect on how shared understanding is achieved, which practices are used, and why they are used.

6. Some words about explicit shared understanding

We keep the discussion of explicit shared understanding rather short in this paper, because the creation and interpretation of explicit specifications, which comes with explicit shared understanding, is rather well understood today. The only crucial point that needs to be stated here is the relationship between explicit specifications and explicit shared understanding.

It is very important to know that the mere existence of explicit specifications (as well as of any other artifact) doesn't imply flawless explicit shared understanding. This is the reason why all specifications and other artifacts need to be *validated*. Validation, typically using the practices listed in Table 3, aims at establishing explicit shared understanding between (and among) stakeholders on the one side and software engineers on the other side. Only when an explicit specification has been validated thoroughly, we can say that this specification constitutes explicit shared understanding.

7. A roadmap for shared understanding

7.1 Where are we today with respect to shared understanding?

Today, as stated in Sect. 1, we make use of shared understanding in daily software engineering life without much reflection about it. Creating and validating explicit specifications where we rely on explicit shared understanding is rather well understood, particularly due to the progress made in requirements elicitation in the last 25 years. In contrast, we neither understand implicit shared understanding well nor do we handle it in a systematic and reflected way, thus underusing the power of implicit shared understanding.

Also today's development processes tend towards the extreme with respect to shared understanding: traditional sequential processes try to make all understanding explicit with extensive documentation, while agile processes aim at using as little documentation as possible, thus strongly relying on implicit shared understanding.

7.2 What can we do and where can we go with existing technology?

The notion of a risk-based, value-oriented approach to specifying quality requirements described in [Gl08] can be extended to requirements in general. That means that for every individual requirement, we determine how to express and represent this requirement so that it yields optimal value, using an assessment of the risk as a guideline. Thus we deliberately decide where we write explicit specifications and where we rely on implicit shared understanding. A similar approach could be chosen for determining which architectural decisions should be documented explicitly and for which ones implicit shared understanding suffices.

As a general rule, we should rely on implicit shared understanding whenever we can afford it with respect to the risk involved. This requires processes that allow frequent, rapid feedback as we have it in today's agile processes. On the other hand, agile-addicts, who advocate producing code and tests as the only explicit artifacts, should note that in most real-world projects, we have high-risk requirements and architectural decisions that need to be documented explicitly in order to keep the risk under control.

As another general rule, systematic assessment of implicit shared understanding needs to be established as a standard practice in the same way as validating explicitly specified requirements is a standard practice today. With the exception of measuring implicit shared understanding, the required assessment practices exist (cf. Table 3).

7.3 Where do we need more research and insight?

The work presented in this paper is based on an analysis of our own experience accumulated over many years, as well as on experience reported in the literature. However, most of this experience is punctual and, with respect to strict scientific criteria, anecdotal.

More research and investigation is needed to come up with analyses and rules that are based on dependable empirical evidence.

Measuring implicit shared understanding is an under-researched topic today. What we have today (cf. the last two rows of Table 3) is rather punctual or preliminary. Any progress in this field would be highly welcome and relevant for industrial practice.

The techniques we are currently using for assessing the risks of requirements are mainly qualitative and approximative. Any progress towards measuring or better estimating such risks would also be highly significant.

Finally, we are short of specific practices that are optimized for specific project settings. An example of such a practice is handshaking [FGB10] which is designed for use in software product management where there is a single product or feature owner and a defined team of software engineers. Having such specific practices for other frequently occurring settings would constitute a significant progress.

8. Related work

There is a large body of existing work on particular problems of shared understanding. A comprehensive discussion of this work is beyond the scope of this paper. However, to the best of our knowledge, nobody so far has attempted to give an overview of the problem, summarize existing practices and shed some light on implicit shared understanding, which are the main topics of this paper.

Further there are large bodies of work in related fields such as requirements elicitation, knowledge management, ontologies, and the semantic web that we neither can survey nor summarize within the scope of this paper.

We just give a few selected pointers to related work here. McKay [MK98] proposes a technique called cognitive mapping for achieving shared understanding of requirements. Hill et al. [HSD01] try to identify shared understanding by analyzing the similarity of documents produced by team members, based on latent semantic analysis. Puntambekar [Pu06] investigates the role of collaborative interactions for building shared knowledge. Gacitua et al. [GMN09] review the role of tacit knowledge in Requirements Engineering. Zowghi and Coulin [ZC05] survey the field of requirements elicitation. This topic is also covered in almost any textbook on Requirements Engineering. Guarino et al. [GOS09] give an introduction to ontologies. Stapel [St12] contributes a theory of information flow in software development.

9. Conclusions

Summary. Shared understanding is important for efficient communication and for minimizing the risk of stakeholder dissatisfaction and rework in software projects. Achieving shared understanding between stakeholders and development team is not easy. Obstacles need to be overcome and enablers be taken advantage of. We have presented essential practices that enable and build shared understanding and practices that allow assessing it. We also have shed light on the handling of implicit shared understanding, which today is less researched and understood than dealing with explicit shared understanding in the form of explicit specifications. A roadmap has been developed that describes how the current state of knowledge and practice can be improved.

Contribution. Our essay represents a first focused overview of the topic of shared understanding for software projects. It combines a synthesis of insight and experience with concrete advice of how to build and manage shared understanding. The results provide guidance for practitioners and represent a basis for future research.

Future Work. In our own future work, we are planning to conduct a systematic survey of shared understanding in requirements engineering, including a comprehensive literature analysis. Generally, we will continue our quest for requirements specification techniques that provide optimal value in given contexts and situations.

Acknowledgements

We thank the members of the Requirements Engineering Research Group at the University of Zurich for valuable comments on earlier versions of this paper.

References

[Be02] Berry, D.M.: Formal Methods: The Very Idea. Some Thoughts About Why They Work When They Work. Science of Computer Programming 42(1), 2002. 11-27.

[FG10] Fricker, S.; Glinz, M.: Comparison of Requirements Hand-Off, Analysis, and Negotiation: Case Study. In Proc. 18th IEEE International Requirements Engineering Conference (RE'10), Sydney, 2010. 167-176.

[FGB10] Fricker, S.; Gorschek, T.; Byman, C.; Schmidle, A.: Handshaking with Implementation Proposals: Negotiating Requirements Understanding. IEEE Software 27(2), 2010. 72-80.

[Gl08] Glinz, M.: A Risk-Based, Value-Oriented Approach to Quality Requirements. IEEE Software 25(2), 2008. 34-41.

[GlW07] Glinz, M.; Wieringa, R.: Stakeholders in Requirements Engineering. IEEE Software 24(2), 2007. 18-20.

[GMN09] Gacitua, R.; Ma, L.; Nuseibeh, B.; Piwek, P.; De Roeck, A.N.; Rouncefield, M.; Sawyer, P.; Willis, A.; Yang, H.: Making Tacit Requirements Explicit. In Proc. 2nd International Workshop on Managing Requirements Knowledge (MARK 2009), Atlanta, 2009. 85-88.

[GOS09] Guarino, N.; Oberle, D.; Staab, S.: What Is an Ontology? In (Staab, S.; Studer, R., eds.): Handbook on Ontologies, International Handbooks on Information Systems, 2nd edition, Berlin: Springer, 2009. 1-17.

[GW89] Gause, D.W.; Weinberg, G.M.: Exploring Requirements: Quality Before Design. New York: Dorset House, 1989.

[HSD01] Hill, A.; Song, S.; Dong, A.; Agogino, A.: Identifying Shared Understanding in Design Using Document Analysis. In Proc. 13th International Conference on Design Theory and Methodology, ASME Design Engineering Technical Conferences, Pittsburgh, 2001.

[MK98] McKay, J.: Using Cognitive Mapping to Achieve Shared Understanding in Information Requirements Determination. Australian Computer Journal 30(4), 1998. 139-145.

[Pu06] Puntambekar, S.: Analyzing Collaborative Interactions: Divergence, Shared Understanding and Construction of Knowledge. Computers and Education 47(3), 2006. 332-351.

[St12] Stapel, K.: Informationsflusstheorie der Softwareentwicklung [A Theory of Information Flow in Software Development (in German)]. PhD Thesis, University of Hannover, Germany, 2012.

[Vi93] Vincenti, W.G.: What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. Baltimore: Johns Hopkins University Press, paperback edition, 1993.

[ZC05] Zowghi, D.; Coulin, C.: Requirements Elicitation: A Survey of Techniques, Approaches, and Tools. In (Aurum, A.; Wohlin, C., eds.): Engineering and Managing Software Requirements. Berlin: Springer, 2005. 19-46.

Design for Future: Das DFG-Schwerpunktprogramm für langlebige Softwaresysteme

Ursula Goltz

Technische Universität Braunschweig
Institut für Programmierung und Reaktive Systeme
Mühlenpfordtstr. 23
38106 Braunschweig
goltz@ips.cs.tu-bs.de

Abstract: Das DFG-Schwerpunktprogramm 1593 - *Design for Future - Managed Software Evolution* wurde gegründet, um fundamentale neue Ansätze in der Software-Entwicklung für den Bereich langlebiger Software-Systeme zu entwickeln. Initiatoren des Schwerpunktprogramms waren Ursula Goltz (Koordinatorin) sowie Gregor Engels, Michael Goedicke, Universität Duisburg-Essen, Wilhelm Hasselbring, Andreas Rausch und Birgit Vogel-Heuser; Ralf Reussner ist Co-Koordinator. Die im Schwerpunktprogramm entwickelten Methoden und Werkzeuge erlauben die Entwicklung von "jungbleibender" Software, die ihre ursprüngliche Funktionalität und Qualität während der gesamten Lebensdauer behält und sich sogar kontinuierlich verbessert. Dazu müssen geeignete Ansätze aus dem Software Engineering mit einem speziellen Fokus auf langlebige Software-Systeme ausgebaut und integriert werden. Die darauf aufbauende Methodik für die Evolution von Software und Software-/Hardware-Systemen ermöglicht die Adaption von Systemen an sich ändernde Anforderungen und Umgebungen. Wir stellen ein neues Paradigma auf, indem einerseits Entwicklung, Anpassung und Evolution von Software und deren Plattformen und andererseits Betrieb, Überwachung und Wartung nicht mehr getrennt, sondern integriert betrachtet werden. Für die Anwendung konzentrieren uns auf zwei konkrete Bereiche: Informationssysteme und - als Anwendung aus dem Bereich Maschinenbau - Produktionssysteme in der Automatisierungstechnik. In Informationssystemen werden große Datenmengen und Anwendungen über einen langen Zeitraum verwaltet. In Produktionssystemen im Bereich Automatisierungstechnik werden langlaufende komplexe Software-/Hardware-Systeme entwickelt und gewartet. Hier besteht die Herausforderung, im laufenden Betrieb effiziente Anpassungen und Verbesserungen durchzuführen. In der ersten Phase des Schwerpunktprogramms (3 Jahre, beginnend ab Sommer 2012) werden 13 wissenschaftliche Teilprojekte gefördert. Insgesamt ist eine Laufzeit von 6 Jahren geplant, das beantragte Gesamtvolumen beträgt 10,1 Millionen Euro.

Wie viel Usability-Engineering braucht das Requirements-Engineering?

Chris Rupp

General Manager
SOPHIST GmbH
Vordere Cramergasse 13
90478 Nürnberg
chris.rupp@sophist.de

Abstract: Software Produkte sind nicht immer einfach und intuitiv benutzbar, was uns Nutzer oft verzweifeln und verärgern lässt. Dies liegt daran, dass wir Endbenutzer zu selten in den Entwicklungsprozess mit eingebunden werden, obwohl die Produkte für uns entwickelt werden. Unsere Ziele und Bedürfnisse werden oft gar nicht betrachtet oder kommen viel zu kurz. Doch was kann man aus Requirements Engineering Sicht tun, um dieser Problematik entgegenzuwirken? Im Rahmen eines Innovationsprojektes beschäftigen sich die SOPHISTen mit der Frage, wie viel Usability Engineering das Requirements Engineering braucht, um die Benutzbarkeit von Softwareprodukten zu stärken und dem User ein positives Nutzungserlebnis zu bieten. Wir suchen nach methodischen Schnittpunkten beider Disziplinen, um anschließend einige Usability-Engineering-Methoden in das Requirements Engineering zu transferieren. Weiter forschen wir an der Erstellung eines Vorgehensmodells, bei dem sich Usability-Engineering-Methoden und Requirements Engineering Methoden eingliedern. Sie bekommen über folgende auch für Requirements Engineers spannende Themen aus dem Usability Engineering etwas zu hören:

• Das Personas-Konzept nach Cooper – und wie es das klassische Stakeholder-Relationship-Management ergänzt

• Die Szenariobasierte Erstellung von Kontextszenarien mit anschließender Erhebung von Anforderungen

• Sowie ein Vorgehensmodell, das Usability-Engineering-Methoden und Requirements Engineering Methoden einordnet

Seien Sie gespannt, welche Zusatzerkenntnisse man aus einem professionellen Usability Engineering als Requirements Engineer ziehen kann.

Synergien nutzen – Personas als neue Anforderungsquelle

In der heutigen Zeit nimmt die Usability (= Benutzbarkeit) von Software eine immer größer werdende Rolle ein. Jeder Benutzer hat den Anspruch, dass Produkte einfach und intuitiv zu bedienen sind, denn nur so arbeitet er mit einem Produkt häufig und vor allem gerne. Dennoch gibt es unzählige Softwareprodukte, die eine schlechte Usability aufweisen und dadurch Benutzer oft verzweifeln lassen. Überwiegend liegt dies daran, dass die Ziele, Bedürfnisse und Verhaltensweisen der Benutzer nicht analysiert werden und diese wichtigen Ergebnisse im weiteren Entwicklungsprozess fehlen. Zudem werden die Disziplinen Requirements-Engineering (RE) und Usability-Engineering getrennt voneinander betrachtet und durchgeführt, sodass es bisher kein Vorgehen gibt, welches die Usability-Aspekte frühzeitig in die Anforderungsermittlung integriert.

Was können wir aus RE-Sicht tun, um diesem Problem entgegenzuwirken?

Wir SOPHISTen haben eine erste Version eines Vorgehensmodels entwickelt, welches Usability-Engineering-Methoden systematisch in das Requirements-Engineering integriert. Dabei sollen sich die Methoden der beiden Disziplinen bestmöglich ergänzen, sodass eine qualitativ hochwertige Anforderungsspezifikation entsteht, die als Basis für den weiteren Entwicklungsprozess zur Verfügung steht. Im ersten Teil unserer Forschungsarbeit haben wir die Ermittlungs- und Dokumentationstechniken beider Disziplinen genauer betrachtet und auf Schnittstellen und Unterschiede hin untersucht. Da sich die genannten Teilbereiche in der Analyse-Phase des allgemeinen Entwicklungsprozesses befinden, war dieses Vorgehen möglich. Bei der Untersuchung stellte sich heraus, dass die Ermittlungstechniken der beiden Disziplinen fast identisch sind. Somit kann das Requirements-Engineering auf Ergebnissen des Usability-Engineerings aufbauen und umgekehrt. Problematisch dagegen schien, dass der Fokus der Ermittlung und Dokumentation jeweils ein ganz anderer ist. Das Requirements-Engineering stellt in der Systemanalyse das System mit seinen Funktionalitäten in den Mittelpunkt und arbeitet eher top-down. Im Usability-Engineering wird dagegen der Fokus auf die Benutzer gelegt und bottom-up vorgegangen. Doch genau diese unterschiedliche Fokussierung eröffnet ein enormes Potential für die kombinierte Variante. So entstehen durch die verschiedenen Schwerpunkte neue Betrachtungswinkel, die als Basis für weitere Anforderungen dienen.

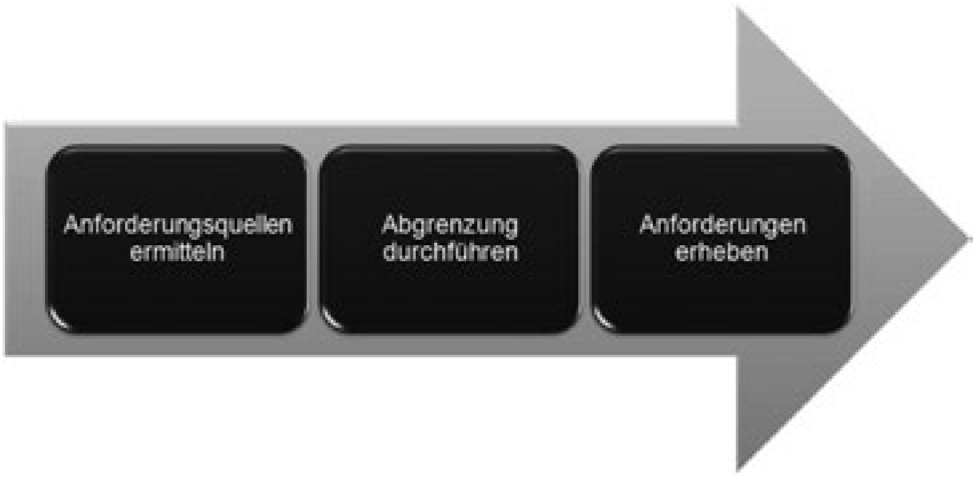


Abbildung 1: Übersicht der methodischen Schnittstellen

Abbildung 1 zeigt drei Schnittpunkte, an denen wir die beiden Disziplinen methodisch kombiniert haben. Die erste Schnittstelle „Anforderungsquellen ermitteln“ zeigt auf, wie sich das klassische Stakeholder-Relationship-Management des Requirements Engineering [Ru09], [PC10] mit dem Persona-Konzept von Alan Cooper bestmöglich ergänzen lässt und wann Personas als neue Anforderungsquellen verwendet werden können. Um im nächsten Schritt eine optimale Kontextabgrenzung durchführen zu können, kombinieren wir die Ergebnisse aus Stakeholder-Interviews und Persona-Beschreibungen. Vor allem die Ziele und das reale Arbeitsumfeld des Benutzers sind dabei von zentraler Bedeutung. Außerdem ist es zu diesem Zeitpunkt manchmal sinniger die potentiellen Anforderungsquellen klar zu priorisieren anstatt später zu viele sich widersprechende Informationen unterschiedlicher Quelle zu konsolidieren. Im letzten Schritt „Anforderungen erheben“ zeigen wir auf, welche Faktoren der Benutzeranalyse als Basis für funktionale und nichtfunktionale Anforderungen dienen und wie diese richtig extrahiert werden.

In diesem Artikel betrachten wir ausschließlich den ersten Schnittpunkt „Anforderungen ermitteln“. Dabei geben wir zuerst einen kurzen Überblick über das Stakeholder-Relationship-Management sowie über die Grundlagen des Persona-Konzeptes, bevor wir Ihnen unser Ergebnis der kombinierten Variante präsentieren.

Was sind Stakeholder und warum sind sie so wichtig?

Im klassischen RE sind Stakeholder Personen oder Organisationen, die Einfluss auf Anforderungen haben. Sie dienen neben Dokumenten und Systemen in Betrieb als wichtigste Anforderungsquelle. Da eine unvollständige Stakeholder-Analyse zu einer lückenhaften Anforderungsspezifikation führen kann, wird versucht, alle Stakeholder mit Hilfe einer Stakeholder-Checkliste zu erfassen. Die ermittelten Stakeholder werden dann in der sogenannten Stakeholder-Liste dokumentiert, die neben essentiellen Attributen, wie Funktion (Rolle) und Wissensgebiet, auch viele organisatorische Details, wie Verfügbarkeit enthält. Da Stakeholder in den gesamten Entwicklungsprozess mit einbezogen werden, sollte die Stakeholder-Liste stets aktuell gehalten werden. Außerdem entscheidet der Projektleiter anhand dieser Liste über die Freigabe von Stakeholder-Ressourcen [Ru09].

Um die Theorie nicht so abstrakt zu halten, verwenden wir folgendes Beispiel: Für eine Restaurantkette soll ein System entwickelt werden, welches die Tätigkeiten des Kellners größtenteils ersetzt (z. B. Bestellung entgegennehmen, Weinempfehlung aussprechen), indem der Gast beim Betreten des Restaurants ein Bediensteil in die Hand bekommt. Mit dem Bedienteil tätigt der Gast z. B. seine Bestellungen, informiert sich über die Herkunft der Lebensmittelund bezahlt die Rechnung. Neben den Tätigkeiten des Kellners soll das System auch administrative Tätigkeiten der weiteren Angestellten systemisch unterstützen (z. B. Lebensmittel bestellen, Urlaub planen).

Im Restaurantbeispiel sind die Stakeholder zum einen die Mitarbeiter des Restaurants (z. B. Geschäftsführer, Koch, Thekenmitarbeiter) sowie alle Gäste. Aber auch Wartungskräfte und Entwickler sind Stakeholder, da sie aufgrund ihrer Funktionen Einfluss auf Anforderungen haben. Nachdem das System verschiedenen gesetzlichen Vorschriften entsprechen muss, wird z. B. Hr. Herold als Repräsentant für das Jugendschutzgesetz mit in die Stakeholder-Liste aufgenommen. Da die Funktion des Kellners durch das System abgelöst werden soll, bleibt er in der Stakeholder-Liste als Anforderungsquelle bestehen. Alle ermittelten Stakeholder werden in der Stakeholder-Liste dokumentiert und anschließend befragt. Abbildung 2 zeigt einen Ausschnitt der Stakeholder-Liste. Bei der Befragung von Stakeholdern im klassischen Requirements-Engineering wird meist der Fokus auf Systemanforderungen und weniger auf die Bedürfnisse und Verhaltensweisen der Benutzer gelegt.

Funktion (Rolle)	Name	Kontaktdaten	Wissensgebiet	Begründung
Geschäfts-führer	Antonio Müller	47mueller@bl.de	Kennt alle Abläufe innerhalb des Restaurants	Entscheidung über Realisierung, ist Geldgeber
Koch	Franco Ferro	ferro@bl.de	Experte für den Lebensmittel-einkauf	Anwender des Systems, muss Lebensmittel bestellen
Entwickler	Franz Huber	huber@4soft.de	Objekt-orientierte Programm-ierung	Technische Realisierung des Systems
Kellner	Tobias Wegerer	tobi-w@aol.de	Speisenservice und Gästebetreuung	Das System soll den Kellner ersetzen.
Rezeption-ist	Michaela Heilmeier	0171 5628452	Reservierungen entgegen-nehmen	Anwender des Systems, für Reservierungen zuständig
Theken-mitarbeiter	Matthias Hansen	0911 3457890	Ausschank, Getränkeservice und Gästebetreuung	Anwender des Systems, für Getränke zuständig
Service-techniker	Thorsten Keller	tk22@gmx.de	Wartung und Reparatur technischer Geräte	Wartung und Reparatur der Endgeräte
Gast	Anna Schweiger	0174 8813401	Administrative Tätigkeiten am PC	Anwenderin des Endgeräts, will Speisen bestellen
Jugend-schutz-beauf-tragter	Benjamin Herold	b.herold@js.com	Experte des Jugend-schutzgesetzes	Jugend-schutzgesetz muss berück-sichtigt werden

Abbildung 2: Ausschnitt einer Stakeholder-Liste

Nachdem wir Ihnen einen kurzen Einblick in das Stakeholder-Relationship-Management gegeben haben, betrachten wir nun eine Methode des Usability-Engineerings genauer. Das Usability-Engineering arbeitet mit dem User-Modell Personas von Alan Cooper [Co07], um die Bedürfnisse der Benutzer zu dokumentieren.

Was sind Personas?

Personas sind User-Archetypen, die die verschiedenen Ziele und beobachteten Verhaltensmuster der potentiellen Benutzer und Kunden beschreiben [Go09]. Bei Personas handelt es sich also nicht um reale Menschen, sondern um Idealtypen, die aber auf den Verhaltensweisen und Zielen echter Menschen basieren. So werden zu Beginn der Systementwicklung Markt- und Designforschung betrieben. Denn bevor ein Produkt den Kundenwünschen nach entwickelt werden kann, müssen erst verschiedene Daten der potenziellen und tatsächlichen Benutzer des Produkts vorliegen. So steht in der sogenannten Research-Phase (Forschungsphase) die Suche nach Bedürfnissen, Ansichten und Zielen der Benutzer im Mittelpunkt [Go09].

Auf das Restaurantsystem bezogen werden die Benutzer des Systems (Mitarbeiter und Gäste) nach ihren Aktivitäten, Zielen, technischen Einstellungen, kognitiven Fähigkeiten, ihrem Umgang mit IT-Systemen, aber auch ihren Ängsten befragt. Besonders wichtig ist es, das mentale Modell der Benutzer – ihre Vorstellung wie das Restaurantsystem funktioniert – zu erforschen. Denn liegt die technische Umsetzung des Restaurantsystems später nahe bei den Vorstellungen der Benutzer, können diese das System intuitiv bedienen. Wichtig bei der Beobachtung und Befragung von Benutzern ist es auch, dass sie in ihrer Umgebung, dem sogenannten Nutzungskontext, betrachtet werden. So wird der Koch bei seinen täglichen Arbeitsaufgaben in der Küche beobachtet und gegebenenfalls dazu befragt. Durch die Kontextbetrachtung können die Ziele der Benutzer besser verstanden werden. Die gesamten ermittelten Informationen werden zusammengefasst und anschließend pro Persona in einem sogenannten Persona-Template visualisiert. Die Abbildung 3 zeigt die Persona „Koch“.

Mit Hilfe von Personas kann über verschiedene Benutzertypen und ihre Bedürfnisse kommuniziert werden, um dann zu entscheiden, welcher für das Design von Form und Verhalten am wichtigsten ist [Go09].

Rolle	Koch	
Name:	Tommaso Zanolla	
Demografische Variablen		
Alter:	34	
Familienstand:	Verheiratet, 2 Kinder	
Verdienst:	Gering	
Wohnort:	Außerhalb der Stadt	
Herkunft:	Italien	
Verhaltensvariablen		

Aktivitäten:	*Einkauf und Lagerhaltung verwalten;* Koch ist dafür zuständig, dass die Woche über genug Lebensmittel und Getränke zur Verfügung stehen. Außerdem ist er auch für die Vorratshaltung verantwortlich. Dazu steht in jedem Restaurant eine Lieferantenakte zur Verfügung.
	Gericht zubereiten; Der Koch bereitet das bestellte Gericht zu.
	Speisekarte verändern; Der Koch muss sich im Abstand von 12 Wochen eine neue Spezialität aneignen. Außerdem kann er bei der Geschäftsleitung die Streichung eines Gerichts beantragen, falls dieses nicht oft bestellt wird.
	Urlaub planen; Urlaubsantrag muss drei Wochen vorher im Kalender der Stammfiliale eingetragen werden.
	Filiale zuweisen; Jeder Mitarbeiter muss sich wöchentlich in der Stammfiliale in die Planungstabelle eintragen und sich dabei einer Filiale zuweisen, in der er die Woche über arbeiten möchte.
	Bestellungen aufgeben; Bestellungen, die nicht Lebensmittel betreffen, werden auf einem vorgegebenen Formular eingetragen.
Einstellungen:	*Technischer Fortschritt*; Der Koch denkt positiv über den technischen Fortschritt und findet die Idee, ein zentrales System zu schaffen sehr gut. Jedoch hat er Angst, dass kurzfristige Bestellungen (Einkäufe) verloren gehen könnten, falls es technische Probleme gibt.
Fähigkeiten:	*Bildungsstand;* Der Koch besitzt eine abgeschlossene Berufsausbildung.
	Fähigkeit zu lernen; Im visuellen Lernen sehr gut.
Ziele:	*Zentrale Einkaufsliste;* Der Koch möchte zu jeder Zeit auf die „Lieferantentabelle“ restaurantübergreifend zugreifen können, um schneller arbeiten zu können.
	Vorratshaltung anzeigen; Der Koch möchte, dass der Vorratsbestand vom System angezeigt wird.
	Alle Spezialitäten anzeigen; Der Koch möchte seine bisherigen Spezialitäten angezeigt bekommen, um sich seiner Kreativität bewusst zu werden.
	Preisvergleich beim Einkauf; Der Koch möchte beim Einkauf einen Preisvergleich der Produkte angezeigt bekommen, um günstiger einkaufen zu können.

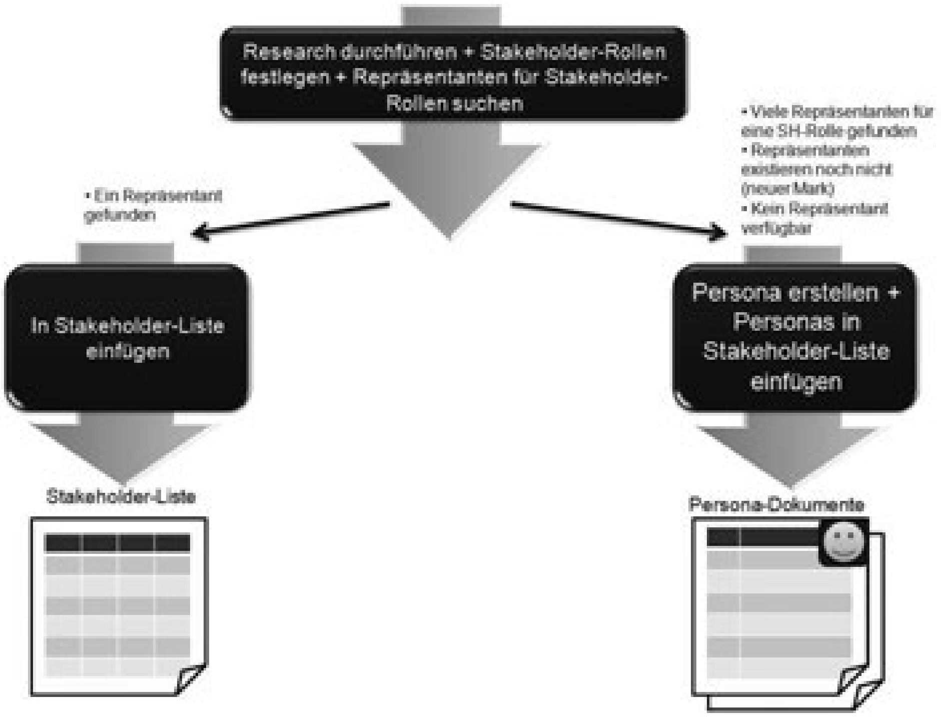
Können:	*Umgang mit dem Computer;* Der Koch besitzt privat einen Computer und nutzt diesen zur täglichen Administration (E-Mails, Word, Excel, Fotos hochladen). Außerdem nutzt der Koch unregelmäßig soziale Netzwerke. Bei kleineren technischen Problemen braucht er keine fremde Hilfe.
Können:	*Umgang mit dem Computer;* Der Koch besitzt privat einen Computer und nutzt diesen zur täglichen Administration (E-Mails, Word, Excel, Fotos hochladen). Außerdem nutzt der Koch unregelmäßig soziale Netzwerke. Bei kleineren technischen Problemen braucht er keine fremde Hilfe.
	Mobiltelefon; Der Koch besitzt ein Smartphone und benutzt dieses öfters als seinen Computer, um ins Internet zu gehen.
Mentales Modell:	*Vorstellung des Systems;* Der Koch hat die Vorstellung, dass die Dokumente in dem System wie ein Stapel Papier aufgebaut sind. In dem Stapel Papier befinden sich alle Tabellen und Listen, die es jetzt in Papierform auch gibt.
Physische Umgebung (Nutzungskont ext):	In der Küche läuft meistens Radio im Hintergrund. Bei verschiedenen Arbeitsschritten ist der Geräuschpegel lauter (Spülen, Braten, Geschirr einräumen, Mixen,…). Außerdem gibt es einige Geräte, die Töne von sich geben (Gefriertruhe, falls zu lange offen, Backofen, nach eingestellter Zeit,…). Der Koch hat in der Küche verschiedene Arbeitsplätze. Diese sind in Lebensmittel- und Schmutzbereich unterteilt. Im Lebensmittelbereich müssen verschiedenste Hygienemaßnahmen durchgeführt werden, um mit Lebensmitteln zu arbeiten. Flüssigkeiten, Fettspritzer und extreme Temperaturen stellen eine Gefahr für Endgeräte dar. Der Koch legt bestimmte Wege in der Küche zurück.

Abbildung 3: Fertige Beschreibung der Persona „Koch“

Nachdem wir Ihnen das Stakeholder-Relationship-Management sowie die Modellierung von Personas vorgestellt haben, kombinieren wir nun die beiden Methoden, um eine vollständige und fundierte Benutzeranalyse zu erhalten.

Wie können sich das Stakeholder-Relationship-Management und Personas ergänzen?

Abbildung 4 zeigt unseren Vorschlag für das kombinierte Vorgehen mit entsprechenden Prozessschritten und Ergebnisartefakten.



Research durchführen + Stakeholder-Rollen festlegen + Repräsentanten für Stakeholder-Rollen suchen
• Ein Repräsentant gefunden
• Viele Repräsentanten für eine SH-Rolle gefunden
• Repräsentanten existieren noch nicht (neuer Markt)
• Kein Repräsentant verfügbar
In Stakeholder-Liste einfügen
Persona erstellen + Personas in Stakeholder-Liste einfügen
Stakeholder-Liste
Persona-Dokumente

Abbildung 4: Methodische Kombination von Stakeholder-Relationship-Management und Personas

Als erstes werden alle Stakeholder-Rollen mit Hilfe der Stakeholder-Checkliste ermittelt und dazu Repräsentanten gesucht. Als weitere Informationsquelle werden hier auch die Techniken des Usability-Engineerings, wie Ethnografie oder Marktforschung eingesetzt, um eine vollständige Stakeholder-Auflistung zu erreichen. Vor allem neue Stakeholder-Rollen aus noch zu erschließenden Märkten können damit ermittelt werden. Wurden Repräsentanten für eine Stakeholder-Rolle gefunden, so werden sie in die Stakeholder-Liste eingetragen. Als Ergebnisartefakt entsteht nach diesem Schritt eine Stakeholder-Liste (vgl. Abbildung 2).

In der Stakeholder-Liste befinden sich nun z. B. zehn Köche und 30 Gäste. Existieren mehrere Stakeholder für eine Funktion (z. B. Koch) oder gibt es eine Stakeholder-Rolle (im Beispiel Gast), die viele gut voneinander abgrenzbare Untergruppen (z. B. Stammgäste oder Laufkundschaft) repräsentiert, werden dafür Personas erstellt. Hier erzeugt eine Persona-Modellierung einen fundierteren Einblick in die Wünsche dieser wichtigen Personengruppe, als das Gespräch mit einem einzelnen Repräsentanten.

Dazu werden die Benutzer, die das Restaurantsystem regelmäßig benutzen, in der Stakeholder-Liste markiert (vgl. Abbildung 5). Das sind Geschäftsführer, Koch, Rezeptionist, Thekenmitarbeiter und Gast. Der Servicetechniker bleibt in diesem Fall außen vor, da er das System nicht so häufig benutzt. In dem Fall, dass es für einen Benutzer genau einen realen Stakeholder gibt, wird dieser nicht zur Persona, sondern wird weiterhin als Stakeholder betrachtet und in der Stakeholder-Liste geführt (z. B. Geschäftsführer). Eine Persona-Modellierung ist unter diesen Umständen überflüssig, da es einfacher ist, den einen realen Stakeholder zu befragen.

Produkt	Stakeholder-Rollen		Begründung
Restaurant-system	Mitarbeiter		Mitarbeiter sind alle Angestellte des Unternehmens und werden somit als eine generalisierte Rolle betrachtet.
		Koch	Koch ist für die Bestellung und Lagerverwaltung der Lebensmittel zuständig.
		Rezeptionist	Rezeptionist hat mehr Befugnisse als die anderen Mitarbeiter und ist vor allem für die Reservierungen zuständig.
		Thekenmitarbeiter	Thekenmitarbeiter führt bestimmte Servicetätigkeiten aus.
	Gast		Gast nimmt Produkte und Dienstleistungen des Restaurants in Anspruch und wird somit als eine generalisierte Rolle betrachtet.

Abbildung 5: Persona-Einteilung abgeleitet aus Stakeholder-Liste

Existieren keine Repräsentanten für eine Stakeholder-Funktion, da sie erst noch geprägt werden müssen (z. B. zukünftiger Markt oder zukünftige Benutzersicht), oder sind die Repräsentanten sehr schwierig zu erreichen, so werden in diesen Fällen ebenfalls Personas erstellt. Nachdem entsprechende Personas erstellt wurden, werden diese mit in die Stakeholder-Liste aufgenommen. Als Ergebnisartefakte existieren nun eine Stakeholder-Liste und mehrere Persona-Beschreibungen. Beide Dokumente behalten ihre ursprüngliche Darstellungsform bei.

Im weiteren Verlauf sollte darauf geachtet werden, dass Stakeholder und Personas gleichwertig für die Ableitung von Anforderungen betrachtet werden. Denn die Gefahr einer starken Fokussierung auf die Benutzer, die im Rahmen der Stakeholder-Analyse gefunden wurden, ist gegeben.

Abschließend können wir sagen, dass durch die Kombination von Stakeholder-Relationship-Management und Personas eine vollständige Benutzeranalyse entsteht, die als solide Basis für die weitere Produktentwicklung dient. Die beiden Disziplinen ergänzen sich dabei in ihrer Vorgehensweise perfekt, indem das Usability-Engineering an vielen Stellen immer wieder die Kreativität des Analytikers anregt und das Requirements-Engineering gleichzeitig die nötige Struktur liefert. Speziell die Benutzeranalyse liefert essentielle Daten wie Ziele, Verhaltensweisen und Bedürfnisse der Benutzer, die im klassischen Requirements-Engineering nicht so detailliert erhoben werden. Die Analyse und Modellierung von Benutzern ist somit eine wichtige Anforderungsquelle für funktionale und nichtfunktionale Anforderungen, da sie bei Entwicklungs- und Designentscheidungen hilft und die Sichtweise auf das System und seine Interaktion erweitert. Die Kombination der beiden Disziplinen erweitert nicht nur den Blickwinkel auf das System, sondern bringt Vollständigkeit für die weiteren Entwicklungsphasen und sichert die Akzeptanz der Nutzer. Außerdem fördert eine Verschmelzung der beiden Disziplinen das Miteinander der Projektteilnehmer langfristig.

Unsere Forschungsarbeit zur methodischen Kombination der Disziplinen Requirements-Engineering und Usability-Engineering ist noch nicht abgeschlossen. So arbeiten wir aktuell an Themenbereichen wie Form- und Verhaltensspezifikation sowie Usability-Tests. Bei Interesse an einer Forschungskooperation zu diesem Thema oder bei Interesse an weiteren Informationen wenden Sie sich bitte per E-Mail an heureka@sophist.de.

Literaturverzeichnis

[Ru09] Rupp, Chris / SOPHISTen, die: Requirements- Engineering und –Management – Professionelle, Iterative Anforderungsanalyse für die Praxis, Hanser, Nürnberg, 2009

[PC10] Pohl, Klaus / Rupp, Chris: Basiswissen Requirements Engineering – Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level, dpunkt.verlag GmbH, Heidelberg, 2010

[Co07] Cooper, Alan / Reimann, Robert / Cronin David: About Face – Interface und Interaction Design, mitp, Heidelberg, 2007

[Go09] Goodwin, Kim: Designing For The Digital Age, Wiley Publishing, Inc., Indianapolis, 2009

Anforderungen an das Software-Engineering in der Automatisierungstechnik

B. Vogel-Heuser[1], C. Diedrich[2], A. Fay[3], P. Göhner[4]

[1]Lehrstuhl für Automatisierung und Informationssysteme, Technische Universität München
vogel-heuser@ais.mw.tum.de

[2]Institut für Automation und Kommunikation e.V., Magdeburg
christian.diedrich@ifak.eu

[3]Institut für Automatisierungstechnik, Helmut-Schmidt-Universität Hamburg
alexander.fay@hsu-hh.de

[4]Institut für Automatisierungs- und Softwaretechnik, Universität Stuttgart
peter.goehner@ias.uni-stuttgart.de

Abstract: Automatisierte Systeme sind komplexe Hardware-Software-Systeme, die zu einem hohen Maß von der Qualität des zugrundeliegenden Software-Engineering abhängen. Der Beitrag soll durch die Klärung der spezifischen Anforderungen zu einem besseren Verständnis von Automatisierungstechnik und Software-Engineering führen und die Grundlagen für eine engere Zusammenarbeit legen. Es werden die wesentlichen Anforderungen identifiziert, die sich aus dem funktionalen Aufbau sowie aus den Randbedingungen des Lebenszyklus automatisierter Systeme ergeben. Auf dieser Grundlage können Methoden und Werkzeuge verbessert und so die Qualität des Software-Engineering gesteigert werden.

1. Software-Engineering in der Automatisierungstechnik

Die Automatisierungstechnik befasst sich mit der Automatisierung von Systemen, die aus Hardware und einem wachsenden Anteil an Software bestehen. Ein automatisiertes System besteht aus einem technischen Prozess, der in einem technischen System abläuft, das alle zur Automatisierung des technischen Prozesses notwendigen technischen Einrichtungen enthält, dem Automatisierungssystem und dem Prozess- und Bedienpersonal [Gö12], siehe Abbildung 1. Ein technisches System ist dabei entweder ein technisches Produkt oder eine technische Anlage. Technische Produkte sind Massenprodukte mit wenigen Sensoren und Aktuatoren und einem hohen Automatisierungsgrad, wie z.B. Haushaltsgeräte. Verglichen mit technischen Produkten sind technische Anlagen Einmalsysteme mit einer großen Anzahl an Sensoren und Aktuatoren und weisen einen mittleren bis hohen Automatisierungsgrad auf, wie z.B. industrielle Produktionsanlagen.

Aus dem speziellen Aufbau der Systeme und den Randbedingungen bezüglich Prozess, Hardware und Interaktion mit dem Prozess- und Bedienpersonal ergeben sich Anforderungen, die im weiteren Beitrag als zusammengehörige *funktional-vertikale Kategorie* behandelt werden.

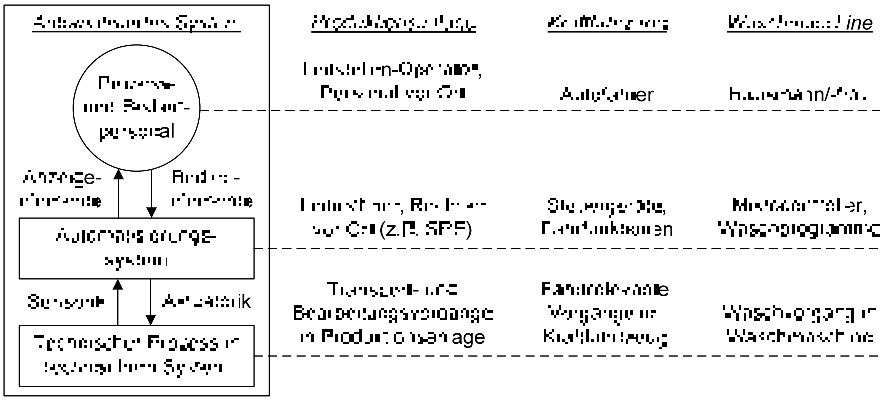

Automatisiertes System
Produktionsanlage
Kraftfahrzeug
Waschmaschine
Prozess- und Bedien-personal
Leitstellen-Operator, Personal vor Ort
Autofahrer
Hausmann/-frau
Anzeige-elemente
Bedien-elemente
Automatisierungs-system
Leitsystem, Rechner vor Ort (z.B. SPS)
Steuergeräte, Fahrfunktionen
Mikrocontroller, Waschprogramme
Sensorik
Aktorik
Technischer Prozess im technischen System
Transport- und Bearbeitungsvorgänge in Produktionsanlage
Fahrrelevante Vorgänge im Kraftfahrzeug
Waschvorgang in Waschmaschine

Abbildung 1: Prinzipieller Aufbau eines automatisierten Systems [Gö12] mit Beispielen

Automatisierte Systeme sind heutzutage im Alltag weit verbreitet. Sie unterstützen ihre Nutzer bei langwierigen, schwierigen oder auch gefährlichen Aufgaben und sind aus dem privaten Bereich, dem Arbeitsleben oder in sich überschneidenden Bereichen, wie dem Individual-, Nah- und Fernverkehr nicht mehr wegzudenken. Aufgrund der Allgegenwärtigkeit im direkten Umfeld der Menschen und der gesellschaftlichen Abhängigkeit von diesen Systemen muss gewährleistet sein, dass die Systeme und insbesondere ihre Software und damit auch das Software-Engineering hohen Qualitätsstandards gerecht werden, um Ausfälle, Fehlfunktionen oder Sach- und Personenschäden möglichst auszuschließen. Allerdings lassen die Kombination von Software und Hardware und die Verbreitung der Systeme in den unterschiedlichsten Anwendungsdomänen die Komplexität der Systeme ansteigen und führen beim Software-Engineering zu komplexen Zusammenhängen, die sich immer schwerer beherrschen lassen [WaRa08]. Zudem erfordert die Entwicklung der Systeme die Zusammenarbeit unterschiedlicher Disziplinen, deren zielgerichtete Koordination entscheidendes Erfolgskriterium ist [Ma10]. Umso wichtiger ist in diesem Zusammenhang ein gemeinsames Verständnis der Anforderungen, die beim Engineering der Systeme zu erfüllen sind.

Beim Software-Engineering in der Automatisierungstechnik ergeben sich entscheidende Anforderungen aus dem gesamten Lebenszyklus der automatisierten Systeme - von Beginn des Engineering, während der Laufzeit, bis zum Ende ihrer operativen Betriebsphase. Diese werden im Weiteren ebenfalls als zusammengehörige *zeitlich-horizontale Kategorie* diskutiert.

In diesem Beitrag werden, ausgehend von den beiden genannten Kategorien, die aus heutiger Sicht bestehenden Anforderungen an das Software-Engineering in der Automatisierungstechnik identifiziert. Diese Anforderungen dienen einem besseren Verständnis und damit einer erfolgreichen und konstruktiven Zusammenarbeit von Automatisierungstechnik und Software-Engineering. Der Beitrag gliedert sich in ein Kapitel zu den grundlegenden Zusammenhängen, deren Betrachtung zu einer Verfeinerung der Kategorien in Subkategorien führt. Im dritten Kapitel werden die konkreten Anforderungen in den Subkategorien detailliert erläutert. Abschließend werden die Anforderungen zusammengefasst und die weiteren Herausforderungen diskutiert.

2. Grundlegende Zusammenhänge in der Automatisierungstechnik

Automatisierung bezeichnet die Überführung von Funktionen eines Systems oder einer technischen Anlage von einem manuellen hin zu einem automatischen Ablauf, um diesen zielgerichtet zu beeinflussen. Der Begriff „automatisch“ bezieht sich hierbei auf den Prozess oder die Einrichtung, „der oder die unter festgelegten Bedingungen ohne menschliches Eingreifen abläuft oder arbeitet“, also selbsttätig erfolgt (vgl. [DIN60050-351], Abschnitt 351-21-40). Nach [DIN60050-351], Abschnitt 351-21-43 ist ein technischer Prozess „ein Prozess, dessen physikalische Größen mit technischen Mitteln erfasst und beeinflusst werden“. Automatisierung erfordert daher im Allgemeinen einen Zugriff auf Informationen aus diesem technischen Prozess (über Sensoren) und/oder eine Möglichkeit des Eingriffs in den technischen Prozess (über Aktuatoren).

Mit Anlagenautomatisierung werden die automatisierungstechnischen Funktionen bezeichnet, wie z.B. auch Sensoren und Aktuatoren, die integraler Bestandteil der Anlagenautomatisierung sind. Der Automatisierungsgrad eines Systems oder einer technischen Anlage entspricht der Norm folgend dem „Anteil der selbsttätigen Funktionen an der Gesamtheit der Funktionen“ des Systems oder der technischen Anlage (vgl. [DIN60050-351], Abschnitt 351-21-41). Bei einem Automatisierungsgrad kleiner eins wird ein System oder eine technische Anlage als teilautomatisiert, andernfalls als vollautomatisiert bezeichnet (vgl. [SS86], S. 375), siehe Abbildung 2.

Ein wichtiges Element bei der Automatisierung ist der Nutzer beziehungsweise die Nutzergruppen, die das System entwickeln, einsetzen, warten und pflegen (Abbildungen 1 und 2). Die *Benutzbarkeit* bildet die aus funktionaler Sicht hierarchisch oberste Subkategorie, in der sich Anforderungen in der funktional-vertikalen Kategorie ergeben.

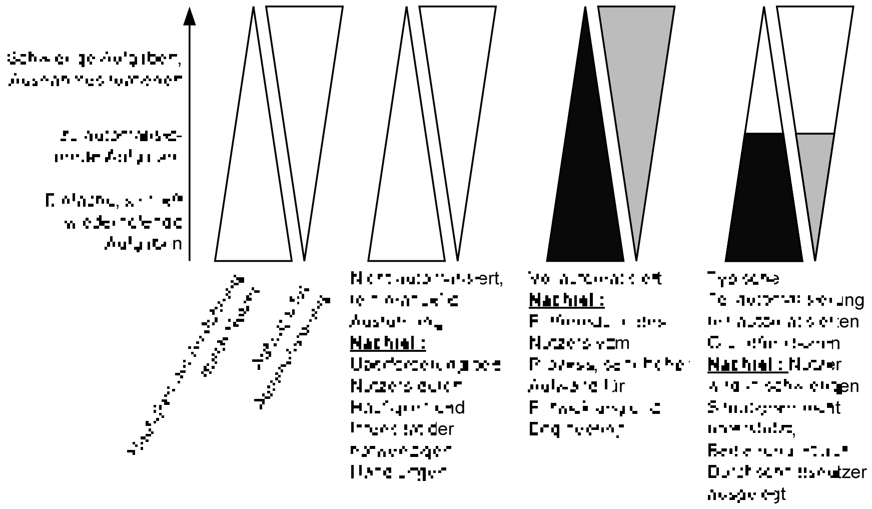


Abbildung 2: Manuelle, voll automatisierte und teilautomatisierte Systeme

Die unterschiedlichen Nutzergruppen beeinflussen den technischen Prozess über die Automatisierungsfunktionen. In Anlagen laufen räumlich verteilt verschiedene Teilprozesse ab (z.B. sequentielle Produktionsschritte). Die dazugehörigen Automatisierungsfunktionen sind daher konzeptionell verteilt und zunächst voneinander unabhängig. Zur Erreichung übergeordneter betrieblicher Ziele (z.B. maximaler Durchsatz) werden übergeordnete Funktionen vorgesehen, welche mit den Automatisierungsfunktionen der einzelnen Produktionsschritte interagieren, was zu weiteren Abhängigkeiten führt.

Die Automatisierungsfunktionen werden heutzutage größtenteils in Software realisiert. Gründe dafür sind die niedrigen Kosten dieser Realisierung und die geringeren Platzanforderungen im Vergleich zu mechanischen oder elektrischen Realisierungen. Vorteilhaft ist bei der Realisierung in Software insbesondere auch, dass die Änderungen und damit Eingriffe während des Betriebs der Anlage kürzere Zeit benötigen als mechanische oder elektrische Umbauten an der Anlage. Außerdem können bei der Signalverarbeitung sehr hohe Genauigkeiten bei ausgesprochener Robustheit gegenüber Umweltbedingungen erzielt werden. Dies ermöglicht auch eine hohe Anpassungsfähigkeit der Funktionen an die spezifischen Bedingungen der einzelnen technischen Prozessstellen.

Die Automatisierungsfunktionen sind das zentrale Element eines automatisierten Systems und haben wesentlichen Einfluss auf die Anforderungen an das Software-Engineering. Die Anforderungen, die sich an die *Automatisierungsfunktionen* ergeben, bilden die zweite Subkategorie der funktional-vertikalen Kategorie.

Die Automatisierungsfunktionen, die in Software realisiert sind, laufen auf speziellen Plattformen ab. Plattform wird hier als Oberbegriff für Hardware als eingebettetes System mit optional Sensor-, Aktuator-, Eingabe- und Anzeigeelementen sowie der Firmware, der kommunikativen Kopplung und hardwarespezifischen Automatisierungsfunktionen verstanden. Abbildung 3 zeigt ausschnittsweise typische Plattformkomponenten, wie sie z.B. bei Antrieben, Bedienpaneelen oder Steuerungen zu finden sind.

Die Berücksichtigung der Eigenschaften der speziellen, in der Automatisierungstechnik eingesetzten Plattformen und Plattformstrukturen ist entscheidend für das Software-Engineering, da nur dann optimal angepasste und qualitativ hochwertige Automatisierungsfunktionen realisiert werden können. Die Anforderungen, die sich durch die *Plattformen* ergeben, bilden die dritte Subkategorie der funktional-vertikalen Kategorie.

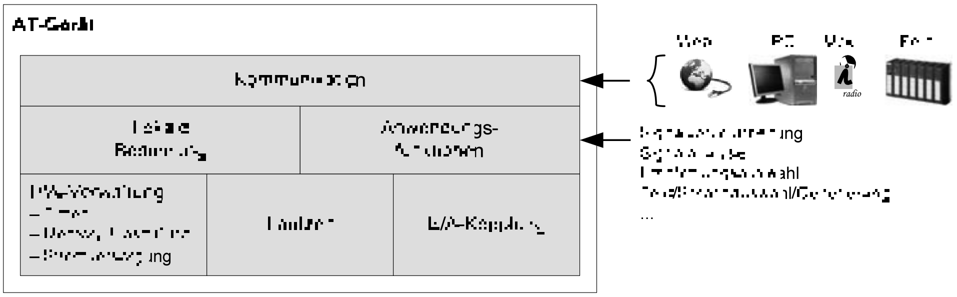


Abbildung 3: Typische Plattformkomponenten eines AT-Geräts.

Betrachtet man schließlich noch die funktional-vertikale Hierarchie als Ganzes, erkennt man, dass sich ein geschlossener Kreis von Abhängigkeiten zwischen Informationserfassung, der Informationsverarbeitung durch die Automatisierungsfunktionen, der Steuer- und Regelsignale der Aktuatoren und der Wirkung im technischen Prozess ergibt. Die Berücksichtigung dieses geschlossenen Kreises, das heißt möglicher Rückkopplungen und Schwingungspotentiale des Systems beim Software-Engineering, ist enorm wichtig, um ein kontrollierbares Verhalten des Systems zu erreichen. Die Anforderungen, die sich durch diesen *geschlossenen Kreis* ergeben, bilden die vierte und letzte Subkategorie in der funktional-vertikalen Kategorie.

Die Definition und Spezifikation neuer Produkte involviert Informationen aus verschiedenen Engineering-Phasen [NA03], z.B. Entwicklung/Konstruktion sowie Betrieb und Wartung ebenso wie aus verschiedenen Engineering-Disziplinen, wie Maschinenbau, Elektrotechnik und Softwaretechnik. Änderungen in einer Disziplin haben Einfluss auf andere Disziplinen. Diese Herausforderungen werden durch die unterschiedlichen Lebensdauern bzw. Änderungszyklen der unterschiedlichen Disziplinen, wie der Mechanik (z.B. Maschine oder Gehäuse), der Elektrotechnik (z.B. automatisierungstechnisches Gerät oder CPU) und der Informationstechnik inklusive der Softwaretechnik erschwert. Während in der Anlagenautomatisierung Maschinen und Anlagen häufig 10 bis 30 Jahre betrieben werden, wird die automatisierungstechnische Hardware im Anlagenbau maximal alle 3 bis 5 Jahre ausgetauscht und die Software maximal alle 6 bis 12 Monate (siehe Abbildung 4, Zeiträume für Produktautomatisierung deutlich kürzer).

Diese Herausforderungen und die zeitlichen Abhängigkeiten im Life-Cycle der Systeme müssen bereits im Software-Engineering berücksichtigt werden, um zur Laufzeit entsprechend flexibel zu sein. Die Anforderungen, die sich an das *Life-Cycle-Management* ergeben, bilden die erste Subkategorie in der zeitlich-horizontalen Kategorie.

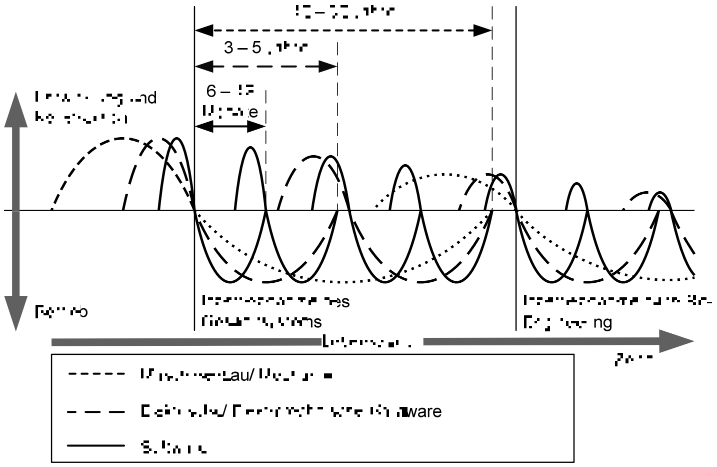


Abbildung 4: Zusammenwirken der verschiedenen Disziplinen in der Anlagen- und Produktautomatisierung in Anlehnung an Li et al. [Li12] (Zeiten für Produktautomatisierung abweichend)

Die besondere Kompetenz der Automatisierungstechniker liegt darin, Methoden aus anderen wissenschaftlichen Disziplinen für die Entwicklung realer technischer Systeme zu einem integrativen Gesamtprozess zu verbinden. Dabei sind im Engineering die besonderen Herausforderungen und Randbedingungen dieser realen technischen Prozesse und Automatisierungssysteme zu berücksichtigen und geeignete und angepasste Methoden erforderlich, um qualitativ hochwertige Ergebnisse zu erreichen.

Dabei basiert die Automatisierungstechnik ganz wesentlich auf Modellen. Modelle werden u.a. für den Entwurf, die Implementierung, den Test, die Optimierung und die Diagnose von Automatisierungssystemen benötigt. In der Automatisierungstechnik werden in der Praxis neben systemtheoretischen Modellen zahlreiche Modelle aus den Fachgebieten verwendet, welche die zu automatisierenden Systeme gestalten. Die automatisierungstechnische Forschung bezieht darüber hinaus Modelle verschiedenster Wissenschaftsdisziplinen, z.B. auch der Informatik, ein und adaptiert diese für die Belange der Automatisierungstechnik. Dabei steht die Automatisierungstechnik immer im Spannungsfeld zwischen der gewünschten Verallgemeinerung der dynamischen Phänomene des zu automatisierenden Systems und der Notwendigkeit, die physikalischen, technischen und auch organisatorischen Gegebenheiten des jeweiligen Anwendungsgebiets zu berücksichtigen, um Modelle, Methoden und schließlich auch Werkzeuge zu entwickeln.

Für den Entwurf und die Realisierung (d.h. Programmierung) von Automatisierungsfunktionen wurden über Jahrzehnte verschiedene Modelle und Beschreibungsmittel entwickelt oder von anderen Disziplinen übernommen und adaptiert. Aus der Mathematik wurden beispielsweise die Boolesche Algebra und die Automatentheorie übernommen, aus der Elektrotechnik der Stromlaufplan (KOP [IEC61131-3]) und die signalorientierte Darstellung der Funktionspläne (FBD [IEC61131-3]). Eine Übersicht über Anforderungen an Beschreibungsmittel zur Steuerungsprogrammierung und eine Bewertung einiger Beschreibungsmittel wurde von der Gesellschaft für Automatisierungstechnik erarbeitet [VDI3681] (siehe Übersicht erweitert um UML-Derivate, Tabelle A1).

Die erforderlichen Eigenschaften der *Modelle* einerseits und der *Werkzeuge* andererseits sind daher entscheidend für das Software-Engineering und bilden die zweite und dritte Subkategorie in der zeitlich-horizontalen Kategorie von Anforderungen.

Die Bemühungen zur Einführung der Automatisierung sowohl in Produktionsprozessen als auch in Prozessen anderer Anwendungsbereiche haben sich in den letzten Jahren grundlegend verändert. Während zunächst die Automatisierung von starren, wiederkehrenden Abläufen im Vordergrund stand, gehen heutige Bestrebungen dahin, flexible Systeme mit unterschiedlichen Aufgabenstellungen einzusetzen. Neben der Installation automatisierter Systeme ist insbesondere auch die Automatisierung von bereits existierenden, jedoch noch nicht automatisierten oder nur teilautomatisierten Prozessen und Abläufen eine wesentliche Aufgabe der Automatisierung. Beispielsweise müssen im Falle von Fehlern in der Anlage Eingriffe in der Software durch Elektriker des Betreibers durchgeführt werden, wie z.B. das Setzen von Ausgängen und die Überbrückung von Sensorabfragen bis zur Behebung eines Hardwarefehlers durch Austausch eines Gerätes (Abbildung 5), um in Betrieb und Wartung den Zeitraum des Anlagenstillstands gering zu halten.

SPS/PLS-Engineering-umgebung aus den Phasen des Engineering	Anforderungsermittlung (Grundlagenermittlung	Nutzeranforderungen			Systemanforderungen	
	Basisplanung					
	Feinplanung/ Konstruktion					
	Subsystemtest im Werk					
	Inbetriebsetzung	Methoden/ Programmteile von Instanzen bilden				
	Abnahme	Sub-System-Test			Systemtest	
	Betrieb und Wartung (Multiuser)	Monitoring der aktuellen Variablen				
		Programm-änderung online	Forcen von Variablen	„Brücken“	Sicherer Austausch von Programmteilen	Änderung von Variablenzuord-nung im Programm
	Re-Engineering: Erweiterung/ Optimierung	Auslesen und Analysieren von bestehenden Programmen			Einspielen geänderter Programmteile	

Abbildung 5: Tätigkeiten im Life-Cycle der Anlagenautomatisierung [Vo09]

Aus diesem Grund ist die Berücksichtigung der Änderbarkeit der Systeme ein wichtiger Faktor im Software-Engineering. Die Anforderungen, die sich an die *Änderbarkeit* ergeben bilden die vierte Subkategorie der zeitlich-horizontalen Kategorie von Anforderungen an das Software-Engineering in der Automatisierungstechnik.

3. Anforderungen an das Software-Engineering

Aus den Bedingungen des Einsatzes von Software in der Anlagenautomatisierung und den grundlegenden Zusammenhängen ergeben sich Anforderungen an die Software, an die sie ausführende Hardware und daraus resultierend, an das Software-Engineering, die in diesem Kapitel, gemäß der Identifizierten Subkategorien gegliedert, erläutert werden.

3.1 Anforderungen an die Benutzbarkeit in der Automatisierungstechnik

Bei der Entwicklung, Nutzung, Wartung und Pflege der Systeme sind viele unterschiedliche Benutzergruppen involviert. Dies führt zu der Anforderung, dass das Software-Engineering all diese unterschiedlichen Personen berücksichtigt und geeignete integrierte Methoden und Prozesse bereitstellt, die ein konsistentes Zusammenarbeiten ermöglichen. Im Einzelnen müssen die folgenden Gegebenheiten berücksichtigt werden:

Im Software-Engineering der Automatisierungstechnik existieren mehrere Ebenen der Software-Erstellung: Zum einen müssen während des Engineering der Automatisierungstechnik die softwaretechnischen Module von Ingenieuren und Informatikern erstellt werden, zum anderen müssen während der Betriebs- und Wartungsphase kurzfristige Änderungen von Technikern und Facharbeitern vorgenommen werden [Vo09].

In den letzten Jahren lässt sich ein Trend im Detail-Engineering erkennen: die Aufteilung in Modulersteller und Modulanwender. Zur Entwicklung von neuen Softwaremodulen für die Modulbibliothek werden häufiger Informatiker eingesetzt, die in der Regel Hochsprachen zur Programmierung bevorzugen und modellbasiert vorgehen wollen.

Für die Erstellung der Applikation werden eher Maschinenbau- oder Elektroingenieure eingesetzt. Diese verwenden vorgefertigte Module und kombinieren diese unter Anwendung der Sprachen der [IEC61131-3]: für die Regelungstechnik Continuous Function

Chart (CFC) und für die Steuerungstechnik in Europa Funktionsbausteinsprache (FBD) bzw. für den Export des Anlagenprojekts ins Ausland im Wesentlichen die Assembler-artige Anweisungsliste (AWL) und der Stromlaufplan-artige Kontaktplan (KOP). Bei anspruchsvollen regelungstechnischen Funktionen bietet sich die Modellierung in MATLAB/Simulink an, die anschließend in IEC 61131-3 Code übersetzt werden muss.

In der Anlagenautomatisierung stehen in der Phase der Wartung und Pflege in Europa in der Regel Facharbeiter oder Techniker zur Verfügung, die auf jeden Fall drei der Programmiersprachen für Speicherprogrammierbare Steuerungen (SPS) beherrschen, die AWL, den KOP und die FBD, während weltweit in der Regel der KOP bevorzugt wird.

3.2 Anforderungen an Automatisierungsfunktionen

Es gibt einen Basiskanon von Automatisierungsfunktionen. Dies betrifft Messen, Stellen und Regeln bzw. Steuern. Bei der generellen Aufgabe der Prozessführung durch die Automatisierung kommen Funktionen wie z.B. Anzeigen, Archivieren, Alarmieren, Beeinflussen und Sichern hinzu. Bei der Umsetzung dieser Funktionen sind Randbedingungen zu beachten, die beispielhaft in Tabelle A2 im Anhang enthalten sind.

Im Vergleich zu Software-Systemen im Bereich Automobil stellen in der Anlagenautomatisierung die Betriebsarten [DIN13128] eine weitere orthogonale Anforderung an die Software-Entwicklung dar. Während diese bei eingebetteten Systemen keine Bedeutung haben, weil diese im Fehlerfall stillgesetzt werden können, muss in der Anlagenautomatisierung das Betreiberpersonal die Anlage im Handbetrieb wieder in einen Zustand fahren, von dem aus diese in den Automatik-Betrieb übergehen und weiter produzieren kann. Dies bedeutet für die Software, dass viele Softwarebausteine nicht nur im Automatik-, sondern auch im Handbetrieb ggf. mit anderen Meldungen, anderen Bedienelementen und reduzierten Geschwindigkeiten betrieben werden müssen und dies in die Software-Entwicklung einbezogen werden muss [FFV12]. Eine Herausforderung der Zukunft ist zudem die geeignete Unterstützung des Operators im Fehlerfall.

Beim Entwurf von Automatisierungs-Software sind zudem weitere Systemaspekte zu berücksichtigen. Die geforderten Automatisierungsfunktionen müssen (im Sinne funktionaler Anforderungen) realisiert werden (dazu gehören bei Produktionsprozessen auch die geforderten Eigenschaften des herzustellenden Produkts). Außerdem müssen die kausalen und zeitlichen Verkopplungen, die über den durch die Automatisierung beeinflussten technischen Prozess und die räumliche Struktur des technischen Systems bestehen (durch Materie- und Energieflüsse sowie Informationen, die diese Flüsse beeinflussen), berücksichtigt werden, um nicht unerwünschte Seiteneffekte oder gar Instabilitäten im System zu erzeugen Des weiteren müssen die Eigenschaften derjenigen Komponenten des technischen Systems, die für die Ausführung von Automatisierungsfunktionen unmittelbar relevant sind (Sensoren, Aktuatoren, informationsverarbeitende Komponenten, Kommunikationseinrichtungen) beachtet werden, zum Beispiel hinsichtlich Signallaufzeit, Bearbeitungsgeschwindigkeit, Rechen- und Speicherkapazitäten, Sicherheit und Zuverlässigkeit. Außerdem müssen die automatisierungstechnischen Funktionen eine hohe Anpassungsfähigkeit durch Konfiguration und Parametrierung ermöglichen, um eine wirtschaftliche Stückzahl der Automatisierungskomponenten zu erzielen.

3.3 Anforderungen durch spezifische Plattformen

Die erfolgreiche Umsetzung der fließenden Automatisierung hängt im Wesentlichen davon ab, wie vorhandene, nachgerüstete und neue Anlagen, Maschinen und Komponenten in ein konsolidiertes Automatisierungskonzept überführt werden können. Hierzu muss eine Softwarelösung bereitgestellt werden, die diese Systeme integriert und die Interoperabilität [DLH11] der vorliegenden heterogenen Soft- und Hardwarelandschaft sicherstellt. Im Einzelnen sind die folgenden Gegebenheiten zu berücksichtigen:

Es gibt eine große Vielfalt an Mikroprozessoren und Echtzeitbetriebssystemen. Dabei gibt es keine Vereinheitlichungstendenzen auf diesem Gebiet in der Automatisierungstechnik. Hauptgrund sind die unterschiedlichen funktionalen Umfänge in den Geräten; von einfachen binären Sensoren zu hoch komplexen Antriebsumrichtern und Analysegeräten. Dabei muss die zeitliche Funktionsausführung nach dem Abtasttheorem beachtet werden, da AT-Funktionen oft auf zeitkontinuierliche Systeme wirken. Es gibt parallele synchrone und asynchrone Funktionsausführung in einem Gerät und es müssen Änderungen der Anwendungsfunktionen während des operativen Betriebs möglich sein.

In der Regel erbringen mehrere durch industrielle Kommunikationssysteme gekoppelte Automatisierungsgeräte gemeinschaftlich in einem Abtasttakt eine Funktion, z.B. geschlossener Regelkreis. Hierbei müssen deterministische industrielle Kommunikationssysteme genutzt und eine Synchronisierung über Gerätegrenzen im µs- bis ms-Bereich ermöglicht werden.

Der Zugriff auf die Automatisierungsfunktionen im operativen Betrieb erfolgt parallel von verschiedenen Host-Systemen mit unterschiedlichen zeitlichen und funktionalen Randbedingungen (z.B. Steuerung, Parametrierung und Service oder Diagnose). Daher sind unterschiedliche QoS der Kommunikation in einem Gerät, der konkurrierende Zugriff auf Funktionen von verschiedenen Hosts, die Bedienung unterschiedlicher Funktionalitäten durch Bediener mit unterschiedlichen Rechten und unterschiedlicher Qualifikation und die Bewahrung der Konsistenz der Funktionsparametrierung erforderlich.

Bezüglich der NFA sind vor allem ein geringer Energieverbrauch (im mW-Bereich, z.B. 100mW) für Geräte in bestimmten Einsatzbereichen, zum Teil sehr hohe Anforderungen an numerische Berechnungsgenauigkeiten (z.B. Durchflussberechnung mit 64-Bit-Gleitkommazahlen), die Erfüllung der funktionalen Sicherheit, gemeinsame Kommunikations- und Energieversorgungsleitungen für Geräte und sehr lange Einsatzdauern (10 Jahre und mehr) üblich und erforderlich. Über diese lange Einsatzdauer müssen auch die Nachlieferung von Ersatzteilen und Softwareanpassungen jederzeit möglich sein.

Domänenspezifische Sprachen zur Konfiguration, Parametrierung und Programmierung der AT-Geräte sind entscheidend für den Einsatz in den vielfältigen Anwendungsbereichen. Zudem sind die funktionale Modularisierung und die Hierarchisierung für die Gewährleistung von Determinismus in der Abarbeitung trotz komplexer Zusammenhänge in den teils sehr großen Systemen (mit mehr als 10.000 I/O-Signalen) wesentlich. Zudem sind Technologien zur AT-Geräteintegration und damit auch Interoperabilität der kooperativen Funktionen und sehr langen Gerätelebenszyklen von herausragender Bedeutung [Di09].

3.4 Anforderungen durch den geschlossenen Kreis

Aus der Tatsache, dass in der Anlagenautomatisierung ein technischer Prozess (die technische Strecke) geregelt und gesteuert werden muss, ergeben sich weitere Anforderungen. Sensoren liefern dazu Informationen über physikalische Größen, die von der Automatisierungs-Software laufend erfasst und mit dem Ziel verarbeitet werden, diese (und ggf. andere) physikalischen Größen wiederum über Aktuatoren zu manipulieren.

Aus dieser Struktur ergibt sich unmittelbar die besondere Bedeutung einiger NFA für Automatisierungs-Software: Die Echtzeitfähigkeit ist essenziell, da die physikalische Umgebung nicht warten kann wie ein menschlicher Benutzer. Betriebssicherheit ist eine weiteres Kriterium, da die automatisierten technischen Systeme in der Lage sind, erhebliche Schäden anzurichten. Eine für Außenstehende häufig nicht unmittelbar erkennbare Konsequenz aus der oben beschriebenen Struktur ist, dass die Anforderungserfassung für diese Art von Automatisierungs-Software grundsätzliche Unterschiede zur Anforderungserfassung „üblicher“ Software aufweist. In der Automatisierungstechnik steht nicht das Verhalten der zu entwerfenden Software im Mittelpunkt, sondern das Verhalten des automatisierten Systems. Das ist etwas anderes als die Einbeziehung der Hardware, auf der die Software läuft, es um ein Verhalten geht, das sich durch die Dynamik des beeinflussten Systems und dessen Rückkopplung mit der Automatisierungs-Software ergibt.

Die meisten Anforderungen müssen daher zunächst für das Gesamtsystem erfasst werden, erst dann können die Anforderungen an die Automatisierungs-Software abgeleitet werden. Zum Beispiel bestehen die Anforderungen bei einem Geschwindigkeitsregelsystem für Fahrzeuge zunächst aus Gütemaßen für den geschlossenen Regelkreis (z.B. maximaler Overshoot oder eine maximale Anstiegszeit in der Antwort auf eine sprungförmige Änderung der Sollgeschwindigkeit). Diesen steht die Dynamik des Fahrzeuges gegenüber („Wie groß ist die Trägheit aufgrund der Masse und die Beeinflussbarkeit durch Motor und Bremse?“). Erst aus der Gesamtbetrachtung beider Teile, bei der häufig Modelle der Strecke erstellt und analysiert werden müssen, können die Anforderungen an die Software abgeleitet werden (zum Beispiel, dass ein PID-Regelalgorithmus mit entsprechenden Parametern eingesetzt werden soll oder dass bestimmte Antwortzeiten eingehalten werden müssen). Dieses Vorgehen unterscheidet sich wesentlich von sonst typischen Anforderungserfassungen in der Informatik, da hier Wissen erfasst und methodisch verarbeitet werden muss, das sich nicht durch Kundenbefragung ermitteln lässt, sondern aus der Anwendung automatisierungs- und regelungstechnischer Analyseverfahren entsteht.

3.7 Anforderungen an das Life-Cycle-Management

In der Anlagenautomatisierung insbesondere im Anlagenbau, gilt es den Stillstand der Anlage durch den Austausch der Software so gering wie möglich zu halten bzw. ganz zu vermeiden. Daraus resultiert die Anforderung der Automatisierungstechnik, die Änderungen der Software während der Laufzeit also des Betriebs der Anlage durchzuführen und auch Teile der Hardware, z.B. Sensoren und Aktuatoren oder Ein- und Ausgangsbaugruppen, während der Laufzeit tauschen zu können sowie die Kompatibilität von Software über entsprechend lange Zeiträume zu gewährleisten.

Wie bereits eingangs erläutert, wirken eine Vielzahl unterschiedlicher Komponenten – Hardware wie Software – in der Automatisierungstechnik zusammen. Dabei differieren die Lebens- und Innovationszyklen dieser Komponenten erheblich. Die Zeit in der eine Anlage betrieben wird ist hochgradig abhängig vom Industriezweig [Sc11]. In der Anlagenautomatisierung ist die Sicherstellung der einwandfreien Funktion der eingebauten Komponenten aufgrund des deutlich kürzeren Life-Cycle der Einzelkomponenten – dies gilt insbesondere für Software – eine große Herausforderung. Zusätzlich werden die Systeme in Bezug auf ihre Funktionalität und Preisgestaltung stetig weiterentwickelt.

Das in [ZVEI10] für den Einsatz in der Automatisierung entwickelte generische Life-Cycle-Modell unterscheidet zwischen Typen und Instanzen von Produkten. Ein Typ stellt dabei ein Produkt dar, das spezifische Anforderungen erfüllt und das durch Integration neuer Funktionen oder durch Einführung neuer Technologien weiterentwickelt wird. Somit entstehen neue Versionen mit unterschiedlichen Eigenschaften. Jede gefertigte Einheit eines Typs bildet eine Instanz dieses Typs und kann durch eine eineindeutige Kennung (z.B. Seriennummer) identifiziert werden. Jede Instanz eines Produktes besitzt eine Lebenszeit (Product Life Time), die vom Zeitpunkt ihrer Erzeugung bis zur Entsorgung reicht. Für Softwareprodukte bedeutet dies, dass verschiedene Versionen parallel genutzt und gepflegt werden müssen. Hierzu ist ein entsprechend leistungsfähiges Versionsmanagement erforderlich, das nach klaren Regeln eindeutige Versionskennungen vergibt. Bei der Definition einer solchen Versionsnomenklatur ist insbesondere der Kompatibilität der Komponenten herausragende Bedeutung beizumessen. Ein Kompatibilitätsmodell auf Basis der [DIN62402] ist in [Sc11, ZVEI10] enthalten. Das Fokussieren aller Beteiligten der Wertschöpfungskette auf das Life-Cycle-Management, die Verstärkung proaktiver Betrachtungen bereits ab der Planungsphase einer Anlage auf der Basis eines gemeinsamen Modells ist eine wesentliche Voraussetzung zur Minimierung der Total Cost of Ownership (TCO).

3.5 Anforderungen an Modelle in der Automatisierungstechnik

Vor diesem Hintergrund wurden in der automatisierungstechnischen Forschung bereits zahlreiche Modell-Beschreibungsmittel entwickelt bzw. aus anderen Wissenschaftsdisziplinen übertragen, adaptiert, weiterentwickelt und kombiniert, so dass eine große Vielfalt entstanden ist.

Die Automatisierungstechnik erfordert folgende Teil-Modelle:

1. Modell des zu automatisierenden Systems, welches seine Struktur und sein Verhalten (Dynamik) beschreibt;
2. Modell der Anforderungen (an Struktur und Dynamik, von funktionalen Anforderungen und nichtfunktionalen Anforderungen);
3. Modelle vorhandener Automatisierungs(teil-)lösungen, die eingesetzt werden sollen;
4. Modelle von Vorgehensweisen zur Konzeption, Ausgestaltung und Realisierung von Automatisierungssystemen;
5. Modelle des automatisierten Gesamtsystems, einschließlich der Überprüfungen, in welcher Weise die Anforderungen erfüllt wurden.

Die Modelle wirken wie in Abbildung 6 gezeigt zusammen: aus den Modellen 1, 2 und 3 entsteht das Modell 5 durch Anwendung der Modelle 4. Diese Modelle können unterschiedlichen Formalisierungsgrad haben: zum Einsatz kommen informelle, semi-formale und formale Modelle. Den Vorzügen formaler Modelle hinsichtlich Analysierbarkeit etc. muss immer der erhebliche Aufwand der Erstellung dieser Modelle für reale Systeme (deren Modelle große Ausmaße annehmen) gegenübergestellt werden.

3.6 Anforderungen an Werkzeuge für das Software-Engineering

Für die Erstellung und Auswertung der Modelle werden geeignete Software-Werkzeuge benötigt. Ein wesentlicher Aspekt ist die Kopplung zu den Werkzeugen der Mechanik, der Elektrotechnik sowie den Werkzeugen in den vorgelagerten Phasen des Lebenszyklus und aufgrund der Notwendigkeit der Änderungen im Betrieb auch der nachgelagerten Phasen. Die im Software-Engineering üblichen Model-Driven Engineering (MDE)-Ansätze mit der Generierung von Code und der ausschließlichen Änderungen im Modell sind in der Anlagenautomatisierung nicht durchsetzbar [VBK11], weil auf den Baustellen weltweit Änderungen im Code auf der Zielplattform schnell durchgeführt werden müssen, eine Rücksprache mit einer im schlimmsten Fall 12 h versetzten Konstruktionsabteilung oder eine Wartezeit aber nicht akzeptabel sind.

Entsprechend der NFA aus dem Lebenszyklus, der Domänenintegration sowie der Nutzer sind folgende Anforderungen an die Werkzeuge hervorzuheben:

Die Werkzeuge müssen über eine lange Zeit verfügbar sein (Reliability → Availability, Maintainability, Portability) und auf die Benutzergruppen zugeschnitten sein (Usability, Functional Suitability). Sie müssen modulare Planungs- und Entwicklungsprozesse unterstützen, da AT-Systeme aus vorgefertigten Systemen zusammengesetzt werden. Deshalb müssen Werkzeuge Bibliothekskonzepte von Komponententypen unterstützen (Maintainability → Reusability) und besonders Konfigurationsprozesse unterstützen. Vor allem für die integrative und erfolgreiche Zusammenarbeit aller Beteiligten ist zudem die Interoperabilität [DFB11] der eingesetzten Software-Werkzeuge und die Unterstützung von Varianten- und Versionsmanagement im Sinne des Life-Cycle Managements unerlässlich.

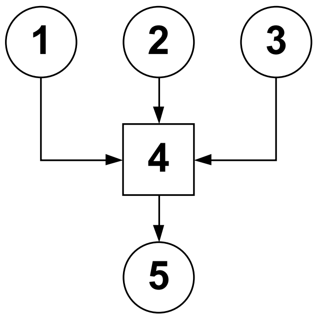


Abbildung 6: Erstellung eines Gesamtsystemmodells für Automatisierungs-Software

3.8 Anforderungen an die Änderbarkeit zur Laufzeit

Änderungen zur Laufzeit sind insbesondere für verfahrenstechnische Prozesse zwingend. Im Bereich des Maschinen- und Anlagenbaus gewinnt der Subsystemtest im Werk immer mehr an Bedeutung, um Inbetriebnahmezeiten auf der Baustelle zu reduzieren. Die Inbetriebsetzung besteht aus Ein-/Ausgangsprüfung sowie der eigentlichen Inbetriebnahme der Subsysteme bis zum Gesamtsystemtest, der durch die Abnahme abgeschlossen wird. Als wesentliche Funktionalität während der Inbetriebsetzung wird das Bilden von Instanzen von Methoden bzw. Programmteilen aus Sicht des Maschinenbaus angestrebt (bei objektorientierter Programmierung), wenn nicht der Applikationsingenieur selbst die Inbetriebnahme durchführt, weil es sich um einen Prototypen einer Anlage oder Teilanlage handelt. Für die Betriebs- bzw. Wartungsphase sind das Monitoring von aktuellen Variablen und die Onlinemanipulation des Programms, wie das Forcen von Variablen oder auch der sichere Austausch von Programmteilen bzw. das sichere Ändern von Variablenzuordnungen im Programm zu unterstützen. Wesentlich ist dabei auch der Multiuser-Modus. Für das Re-Engineering bzw. die Optimierung der Anlage sind insbesondere das Auslesen und das Analysieren von bestehenden Programmen wesentlich. Die Fähigkeiten der Online-Beobachtung und -Änderungen ist eines der Alleinstellungsmerkmale der SPS im Vergleich zu IPCs und der Hochsprachenprogrammierung. Der finale Test der Software erfolgt in der Regel vor Ort auf der Anlage, da nur dort die Randbedingungen vorhanden sind, um alle Aspekte der Funktion zu prüfen, z.B. die Luftfeuchte, die für viele Prozesse relevant ist, oder das Rohmaterial für die Fertigung.

4. Zusammenfassung und Ausblick

Der vorliegende Beitrag gibt eine erste Einführung der wesentlichen Anforderungen an die Software und das Software-Engineering in der Anlagenautomatisierung. Die Anforderungen werden dabei aus zwei Sichten betrachtet. Zum einen werden die funktionalen Eigenschaften, Randbedingungen und Gegebenheiten betrachtet und aus dem vertikalen Aufbau der Systeme, vom Bediener bis zum technischen Prozess, einzelne Subkategorien von Anforderungen gebildet. Innerhalb dieser Subkategorien – den Anforderungen an die Benutzbarkeit, an die Automatisierungsfunktionen, den Anforderungen durch die verwendeten Plattformen und den geschlossenen Kreis – werden die einzelnen Anforderungen detailliert beschrieben. Zum anderen ergeben sich Anforderungen aus Gegebenheiten zu unterschiedlichen Zeitpunkten in den einzelnen Phasen des Life-Cycle der automatisierten Systeme. Diese Anforderungen werden zeitlich-horizontal in Subkategorien – Anforderungen an das Life-Cycle-Management, an die Modelle, an die Werkzeuge und an die Änderbarkeit zur Laufzeit – unterteilt und ebenfalls detailliert erläutert.

Viele der Ansätze aus dem allgemeinen Software-Engineering, wie die Objektorientierung oder MDE-Ansätze, sind bereits in die Anlagenautomatisierung eingeflossen, stoßen aber an ihre Grenzen, wenn sie nicht an die Anforderungen adaptiert werden. Der Beitrag möchte die Diskussion und die Kooperation zwischen Informatik und Automatisierungstechnik auf Basis eines besseren gegenseitigen Verständnisses verstärken.

Literaturverzeichnis

[DLH11] Diedrich, C.; Lüder, A.; Hundt, L.: Bedeutung der Interoperabilität bei Entwurf und Nutzung automatisierter Produktionssysteme. Automatisierungstechnik (at), Vol. 59, Nr. 7, 2011, S 426–438.

[Di09] Diedrich, C.: Feldgeräte-Instrumentierungstechnologien – Integration intelligenter Feldgeräte in PLS. In: Handbuch der Prozessautomatisierung, München. Oldenbourg Industrieverlag, 2009, S. 290–322.

[DIN13128] DIN EN 13128: Sicherheit von Werkzeugmaschinen – Fräsmaschinen (einschließlich Bohr-Fräsmaschinen). 2009.

[DIN60050-351] DIN IEC 60050-351: Internationales Elektrotechnisches Wörterbuch – Teil 351: Leittechnik. 2009.

[DIN62402] DIN EN 62402: Anleitung zum Obsoleszenzmanagement. 2007.

[DFB11] Drath, R.; Fay, A.; Barth, A.: Interoperabilität von Engineering-Werkzeugen: Konzepte und Empfehlungen für den Datenaustausch zwischen Engineering-Werkzeugen. Automatisierungstechnik (at), Vol. 59, Nr. 3, 2011, S. 451-460.

[FFV12] Fuchs, J.; Feldmann, S.; Vogel-Heuser, B.: Modularität im Maschinen- und Anlagenbau – Analyse der Anforderungen und Herausforderungen im industriellen Einsatz. In: EKA, Magdeburg, 2012, S. 306–317.

[Gö12] Göhner, Peter: Automatisierungstechnik I, Skript zur Vorlesung Automatisierungstechnik I, Institut für Automatisierungs- und Softwaretechnik, Universität Stuttgart, 2012.

[IEC61131-3] IEC 61131-3: Speicherprogrammierbare Steuerungen – Teil 3: Programmiersprachen. 2009.

[Li12] Li, F. et al.: Specification of the Requirements to Support Information Technology-Cycles in the Machine and Plant Manufacturing Industry. In: 14th IFAC INCOM Symposium, Bukarest, Rumänien, 2012.

[Ma10] Maga, C. et al.: Mehr Systematik für den Anlagenbau und das industrielle Lösungsgeschäft – Gesteigerte Effizienz durch Domain Engineering. Automatisierungstechnik (at), Vol. 58, Nr. 9, 2010, S. 524-532..

[NA03] NA 35: Abwicklung von PLT-Projekten. Arbeitsblatt Namur. 2003.

[Sc11] Schrieber, R. et al.: Kompatibilität: der zentrale Schlüssel für Nachhaltigkeit – Life-Cycle-Excellence durch proaktives Handeln. Automatisierungstechnische Praxis (atp), Vol. 53, Nr. 11, 2011, S. 50–57.

[SS86] Spur, G.; Stöferle, T.: Handbuch der Fertigungstechnik – Fügen, Handhaben und Montieren. München, Hanser Verlag, 1986.

[VBK11] Vogel-Heuser, B.; Braun, S.; Kormann, B.: Implementation and evaluation of UML as modeling notation in object oriented software engineering for machine and plant automation. In: IFAC World Congress, Mailand, 2011, S. 9151 – 9157.

[VDI3681] VDI/ VDE 3681: Einordnung und Bewertung von Beschreibungsmitteln aus der Automatisierungstechnik. 2005.

[Vo09] Vogel-Heuser, B. (Hrsg.): Automation & Embedded Systems – Effizienzsteigerung im Engineering. Kassel University Press, 2009.

[WaRa08] Wahlster, W.; Raffler, H. (Hrsg.): „Forschen für die Internet-Gesellschaft: Trends, Technologien, Anwendungen“, Feldafinger Kreis, Feldafing, 2008.

[ZVEI10] ZVEI e.V.: Life-Cycle-Management für Produkte und Systeme der Automation. Frankfurt, 2010, Online: http://www.zvei.org/Verband/Publikationen/Seiten/Life-Cycle-Management-für-Produkte-und-Systeme-der-Automation.aspx.

Anhang

Tabelle A1: Erfüllung der Einordnungskriterien durch ausgewählte Beschreibungsmittel in Anlehnung an [VDI3681]

Kriterium	Formale Basis			Verhaltens-beschreibung				Explizite Zeitdarstellung			Struktur			Synchronisation (falls verteilte Prozesse möglich)			Darstellung			Implemen-tierungs-aspekte				Architektur-beschreibung			Tools		
Beschreibungsmittel	Formal	Semi-formal	Informal	Deterministisch	Nicht deterministisch	Statisch	Dynamisch	Ereignisgetrieben diskret	Zeitdiskret (taktgetrieben)	Zeitkontinuierlich	Hierarchie	Komposition / Dekomposition	Strukturveränderung	Synchron	Asynchron	Nebenläufig	Textuell	Mathematisch-symbolisch	Grafisch	Scheduling	Taskaktualisierung	Tasksynchronisation	Kommunikation zwischen Tasks	Hardwarekomponenten	Kommunikationsbeschreibung	Mapping	Forschung	Prototypische Anwendung	Industrielle Anwendung
Ablaufsprache (AS)		x		x			x	x			x	x				x			x^1										x
agile SW (UML)		x		x^7		x	x	x	x		x	x	x	x	x	x	x	x	x	x			x				x	x	x
Algebraische Modelle	x			x	x	x	x	x	x			x		x	x			x									x		
Anweisungsliste (AWL)		x		x		x	x	x			x	x					x												x
Automaten	x			x	x		x	x	x	x^2	x^3	x		x^4	x		x	x	x								x		x
Blockschaltbild (Regelungstechnik)	x			x			x			x	x	x		x				x	x								x		x
Funktionsbaustein-Sprache (FBS)		x		x		x	x	x	x		x	x							x										x
Funktionsblöcke nach IEC 61499		x		x		x	x	x			x	x			x	x			x^1									x	
Idiomatic Control Language (ICL)		x				x		x	x	x											x	x	x					x	
Kontaktplan (KOP)		x		x		x	x	x			x								x										x
Message Sequence Charts (MSC)		x		x	x		x												x										x
Petrinetze (PN)	x	x		x	x		x	x	x	x	x					x			x^1									x	
Procedural Function Charts (PFC)		x		x			x	x			x	x	x			x			x										x
Programmablaufgraph (PAG)	x			x		x	x												x										x
SA/RT		x				x		x			x	x	x				x		x					x	x	x		x	
Specification and Description Language (SDL)	x			x	x		x	x	x		x	x			x				x										x
Strukturierter Text (ST)		x		x		x	x	x			x						x												x
UML		x			x	x	x				x	x														x	x		x
UML-PA	x			x		x	x	x	x	(x)	x	x	x											x	x	x		x	
UML-RT		x				x	x	x			x	x												x		x	x		x
VHSIC Hardware Description Language (VHDL)	x			x	x	x	x	x	x	x^5	x	x		x	x	x	x		x^6										x

Tabelle A2: Aspekte von Automatisierungsfunktionen (DE-Domänenexperte)

	Hardware-Voraus-setzungen	**Zeitliche Vo-raussetzungen**	**Beschrei-bungsmittel**	**Nutzer Human Machine Inter-face**	**…**
Messen	an Sensor/Mess-umformer ge-bunden, Timer für Laufzeit-überwachung	Abtasttheorem	Eingebettetes System bei Entwicklung	Anlagenfahrer, Betriebspersonal Servicepersonal	
Stellen, Schalten	an Aktuator gebunden, Timer für Laufzeit-überwachung	Abtasttheorem	Eingebettetes System bei Entwicklung	Anlagenfahrer, Betriebspersonal Servicepersonal	
Regeln, Steuern	nicht Hardware gebunden, Timer für Laufzeit-überwachung	Abtasttheorem	programmierbar durch Meister, Techniker, AT- oder DE, z.B. IEC 61131-3	Anlagenfahrer, Betriebspersonal Servicepersonal	
Archivie-ren	nicht Hardware-gebunden	mit Zeitstempel	wie IT Systeme	Anlagenfahrer, Betriebspersonal Servicepersonal	
Regis-trieren	nicht Hardware-gebunden	mit Zeitstempel	wie IT Systeme	Anlagenfahrer, Betriebspersonal Servicepersonal	
Anzeigen	Industrie-tauglichkeit	in menschlicher Dynamik (1/2 ms)	konfigurierbar durch Meister, Techniker, AT- oder DE	Anlagenfahrer, Betriebspersonal Servicepersonal	
Beein-flussen	Industrie-tauglichkeit	in menschlicher Dynamik (1/2 ms)	konfigurierbar durch Meister, Techniker, AT- oder DE	Anlagenfahrer, Betriebspersonal Servicepersonal	
Sichern	Besondere Anforderungen	Abtasttheorem	Spezialisten, TÜV, IEC 61508	Anlagenfahrer, Betriebspersonal Servicepersonal	

Die Saarbrücker Graduiertenschule der Informatik

Andreas Zeller
Universität des Saarlandes
zeller@cs.uni-saarland.de

Die im Rahmen der Exzellenzinitiative geförderte *Saarbrücker Graduiertenschule der Informatik* umfasst die gesamte Doktorandenausbildung in der Informatik in Saarbrücken. Rund 340 Doktoranden arbeiten bei etwa 180 etablierten Informatikwissenschaftlern, verteilt auf mehrere große, miteinander verknüpfte und eng kooperierende Forschungseinrichtungen, die alle direkt nebeneinander auf dem Saarbrücker Campus der Universität des Saarlandes beheimatet sind. Alle Doktoranden absolvieren das Programm, das nach einem harten Auswahlverfahren den direkten Weg vom Bachelor zur Promotion mit Finanzierungsgarantie ermöglicht, und Doktorandenausbildung einschließlich Zulassung und Fortschrittskontrolle als Gemeinschaftsaufgabe aller Professoren betrachtet.

Mit der Graduiertenschule hat sich die Saarbrücker Informatik ehrgeizige Ziele gesetzt, die Informatik als Grundlagenwissenschaft zu pflegen und auszubauen. Die Informatik in Deutschland – und ganz besonders die Softwaretechnik! – muss sich entscheiden, ob sie als Hilfswissenschaft Ingenieuren und Betriebswirtschaftlern zuarbeiten möchte, oder ob sie aus sich heraus eigenständige Erfolgsgeschichten feiern kann und will.

Andreas Zeller ist seit 2001 Professor für Softwaretechnik an der Universität des Saarlandes in Saarbrücken. In 2010 wurde Zeller zum *Fellow der ACM* ernannt für seine Beiträge zur automatischen Fehlersuche und der Analyse von Software-Archiven. In 2011 erhielt er einen *ERC Advanced Grant*, Europas höchste und bedeutendste Forschungsförderung, für Arbeiten über Specification Mining und Testerzeugung. Seit 2012 ist er gewählter Fachkollegiat der DFG für das Fach „Softwaretechnologie“.

SE | 13 SOFTWARE ENGINEERING

Forschungsberichte

SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems[*]

Matthias Becker, Steffen Becker
Heinz Nixdorf Institute
University of Paderborn
{matthias.becker, steffen.becker}@upb.de

Joachim Meyer
SAP AG
joachim.meyer@sap.com

Abstract: Modern software systems adapt themselves to changing environments in order to meet quality-of-service requirements, such as response time limits. The engineering of the system's self-adaptation logic does not only require new modeling methods, but also new analysis of transient phases. Model-driven software performance engineering methods already allow design-time analysis of steady states of non-adaptive system models. In order to validate requirements for transient phases, new modeling and analysis methods are needed. In this paper, we present SimuLizar, our initial model-driven approach to model self-adaptive systems and to analyze the performance of their transient phases. Our evaluation of a proof of concept load balancer system shows the applicability of our modeling approach. In addition, a comparison of our performance analysis with a prototypical implementation of our example system provides evidence that the prediction accuracy is sufficient to identify unsatisfactory self-adaptations.

1 Introduction

Modern business information systems run in highly dynamic environments. Dynamics range from unpredictably changing numbers of concurrent users asking for service, to virtualized infrastructure environments with unknown load caused by neighboring virtual machines or varying response times of required external services. Despite such dynamics, these systems are expected to fulfill their performance requirements. In the past designers achieved this by overprovisoning hardware, which is neither cost-effective nor energy-preserving. Self-adaptation is a primary means developed over the last years to cope with these challenges. The idea is that systems react to their dynamic environment by restructuring their components and connectors, exchanging components or services, or altering their hardware infrastructure.

To deal with performance requirements of classical, non-adaptive systems, researchers developed model-driven software performance engineering approaches [CDI11]. These approaches allow early design-time performance predictions based on system models to validate performance requirements. However, classical performance engineering approaches

[*]**Acknowledgments** This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

use non-adaptive system models that do not support a dedicated self-adaptation viewpoint. Consequently, self-adaptive behavior is not considered in the prediction, and only the performance of a single system configuration, i.e., steady states, can be predicted.

The limitation on a steady state performance prediction also limits the range of analysis. For example, consider a web server system with multiple servers which is able to adapt its load balancing strategy to its actual workload. Whether this system is able to recover from an overload situation within an acceptable time or not cannot be answered when neglecting the transient phases, i.e., the self-adaptation phase in which it switches from one load balancing system configuration to another configuration. Neither can it be answered if the workload is balanced over just as many servers as really needed and is hence cost-efficient.

The contribution of this paper is SimuLizar, a modeling and model-driven performance engineering approach for self-adaptive systems. SimuLizar is based on the Palladio approach [BKR09]. It extends Palladio's modeling approach with a self-adaptation viewpoint and a simulation engine. The latter enables the performance prediction of self-adaptive systems over their various configurations, allowing the analysis of transient adaptation phases.

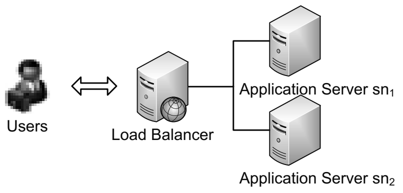
To evaluate our approach, we have applied our modeling approach to a proof of concept load balancer system. The performance analysis of the system's self-adaptation logic shows sufficiently similar characteristics as measurements taken from a performance prototype and allows us to identify unsatisfactory self-adaptation logic.

The remainder of this paper is structured as follows. We first provide a specification for a small self-adaptive load balancer in Section 2 as a motivating example. We use this load balancer system to illustrate and to evaluate the applicability of our approach. In Section 3, we briefly introduce the foundations of our work. Our SimuLizar approach is detailed in Section 4. We evaluate and discuss SimuLizar in Section 5. In Section 6, we compare our approach to related work. Finally, we conclude our work and discuss future work in Section 7.

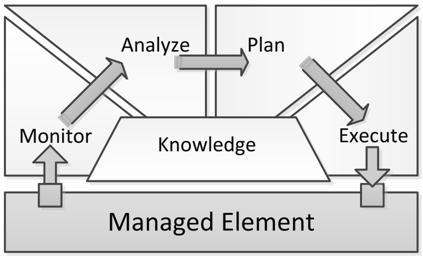
2 Motivating Example

We provide requirements and initial design ideas for a small load balancer example system to which we will refer to throughout this paper. The load balancer we describe is not a real-life example, but serves as an understandable and easily implementable running example of a self-adaptive system.

Requirements. We want to design a load balancer system, as illustrated in Figure 1a, that distributes workload across two single-core application servers sn_1 and sn_2. The load balancer accepts client requests and delegates them to one of two application servers. Responses are sent directly from application servers to the clients. A request causes constant load of 0.3 seconds uncontended CPU time on the application server it is delegated to. We assume that we have to pay an additional load-dependent fee for utilizing the second application server sn_2. Even though the assumption of being charged load-dependently is not common for cloud services yet, our industry partners predict that this will be an option in the near future.



(a) Motivating example: load balancer



(b) MAPE-K feedback loop [Mur04]

Figure 1: Motivating example and MAPE-K feedback loop.

The following requirements (R1-R3) shall be fulfilled by the load balancer system:

- **R1** The system must keep response times for user requests low. The response time for a user request must not be greater than 0.8 seconds in the mean.
- **R2** The system must keep the infrastructure costs as low as possible.
- **R3** In case R1 is not fulfilled, i.e., the mean response time is greater than 0.8 seconds, the system has to re-establish a mean response time of 0.8 seconds or less as fast as possible.

Initial Design. A software architect wants to design the load balancer such that it works properly even in high load situations. She provides initial design ideas and documents design decisions.

In order to fulfill R1, the software architect could design a load balancer to randomly delegate user request to the application servers sn_1 and sn_2. Since this conflicts with R2 the load balancer needs to preferably delegate user requests to application server sn_1 as long as R1 holds.

It is also required by R1 that the response time is less than 0.8 seconds in the mean. As defined in R3 and to save costs, only in case that requirement R1 cannot be fulfilled the load balancer delegates user request to the second application server sn_2. Regardless, it must not delegate more user requests to application server sn_2 than to application server sn_1.

In order to decide whether the mean response time is greater than 0.8 seconds, the system needs to measure the response time for each request and calculate the mean. However, it is not specified over which time span the mean should be calculated. This is again a design trade-off. Choosing a longer time span means outlier measurements are more likely ignored; choosing a shorter time span means the system detects earlier that the mean response time is greater than 0.8 seconds but also increases the monitoring overhead. The software architect chooses to calculate the mean response time from response batches within short time spans of 20 seconds. However, the software architect cannot predict whether R3 can be fulfilled yet, i.e., it cannot be predicted if the system recovers from high load situations fast enough.

In the following section, we introduce the foundations of self-adaptive systems and model-driven software performance engineering (MDSPE). Based on our example, we will create a formal system model from our initial design ideas in Section 4.

3 Foundations

First, we outline the characteristics of self-adaptive systems. Second, we introduce model-driven software performance engineering (MDSPE) and the Palladio MDSPE approach on which SimuLizar is based. We will refer to these foundations when we introduce our modeling approach and our Palladio-based performance engineering approach for self-adaptive systems in Section 4.

Modeling Self-Adaptive Systems. Self-adaptive systems are able to adapt their structure, behavior, or allocation in order to react to changing environmental situations. In our research, we focus on self-adaptive systems based on the MAPE-K feedback loop [Mur04], as illustrated in Figure 1b. The MAPE-K feedback loop consists of four steps. First, the managed element, the self-adaptive system, is *monitored* via defined sensors, e.g., for the mean response time. Second, the monitored data is *analyzed*, e.g., if the predefined threshold of 0.8 seconds is exceeded. Third, if the analysis revealed that a self-adaptation is required it is *planned*, e.g., a predefined self-adaptation rule like the load balancing rule is selected. Finally, the self-adaptation is *executed* via effectors of the managed element. In all steps, a *knowledge base* containing system information, i.e., a runtime model, can be accessed. For example, monitoring data can be stored in the knowledge base, or self-adaptation rules can be accessed from the knowledge base of the MAPE-K feedback loop.

The design and analysis of these systems is made difficult by two factors. First, the broad variety of environmental situations to which self-adaptive systems can adapt, e.g., actual workloads, requires a complex logic to monitor and analyze the environment. Second, the wide range of possible self-adaptation tactics, e.g., adapting the load balancing strategy in our motivating example, introduces additional complexity for planning and executing self-adaptive behavior.

There is an ongoing trend in software engineering to introduce a new modeling viewpoint in order to address the increased complexity of self-adaptivity [Bec11]. This new modeling viewpoint enables a separation of concerns, i.e., separating business logic from self-adaptation logic by introducing new views for modeling monitoring and reconfiguration. Thus, a dedicated analysis of requirement fulfillment of the self-adaptation logic is enabled. Our SimuLizar approach enables us to model self-adaptive systems with this new modeling viewpoint. In the next section, we provide the foundations of the performance analysis as extended by SimuLizar.

Model-Driven Software Performance Engineering. Model-driven software performance engineering is a constructive software quality assurance method to ensure performance-related quality properties [CDMI06]. Software performance properties can be quantified using several metrics, like response-time, or utilization.

Assuming that performance requirements are explicitly specified, MDSPE enables to evaluate whether performance requirements can be satisfied or not by a modeled software system. For this purpose a software design model is annotated with performance-relevant resource demands, such as CPU time demands. For example, in our load balancer system we know that a user request has a constant CPU demand of 0.3 seconds.

Subsequently, the annotated model is translated into analysis models or a performance prototype. Analysis models can either be simulated or solved using analytical methods, e.g., solving queuing networks with queuing theory. Performance prototypes can be deployed to the target runtime environment, i.e., servers, and performance metrics like response times can be actually measured. A correct transformation ensures that the transformed model or performance prototype is a correct projection of the software design model.

Which method of analytical solving, simulation, or performance prototyping is applied is mainly a trade-off among made assumptions and the accuracy of the performance prediction. In general, analytical solving provides accurate predictions if strict assumptions hold, e.g., the model must only include exponentially distributed processing rates and inter-arrival rates in order to be solvable. Simulation allows more relaxed assumptions but provides accurate predictions only with a high number of simulation runs. Since a performance prototype is deployable and runnable software that fakes resource consumption, the fewest assumptions have to hold for it. However, because a performance prototype needs to be deployed and executed on the target execution environments, i.e., real servers, is the most time-consuming of all performance prediction methods.

Finally, the results from analytical solving, simulation, or the performance prototype shed light on whether the performance requirements can be satisfied or not. Interpreting the results also helps revise the software design and eliminating performance flaws.

Palladio [BKR09] is an MDSPE approach for component-based software engineering that our approach is based on. One of Palladio's key features is the integrated support for design and performance analysis of component-based software. For this purpose, Palladio introduces its own component model, the Palladio Component Model (PCM), which allows us to annotate performance-relevant information in a software design model.

A PCM model consists of several artifacts, as illustrated in Figure 2. Component developers provide software components including performance-relevant properties and behavior in a *repository*. A software architect defines the *assembly* by combining components provided in the repository into a complete software system. A deployer specifies the available resource infrastructure, e.g., servers with CPUs and HDDs in a *resource environment* view. Furthermore, she specifies the concrete deployment of the system in an *allocation* view. Finally, domain experts specify typical workloads in a *usage* model.

Performance metrics, e.g., response time, of systems modeled as PCM instances can be analyzed using the Palladio tool suite. For this purpose the PCM model is automatically transformed to simulation code, performance prototypes, or analysis models.

In this paper, we focus on the two tools SimuCom [BKR09] and ProtoCom [BDH08] included in the Palladio tool suite. SimuCom is Palladio's simulation engine, which enables the simulation and performance analysis of systems modeled with PCM. Our SimuLizar approach reuses the functionality provided by the SimuCom engine and extends it to simulate self-adaptive systems. ProtoCom is a tool to transform PCM instances into perfor-

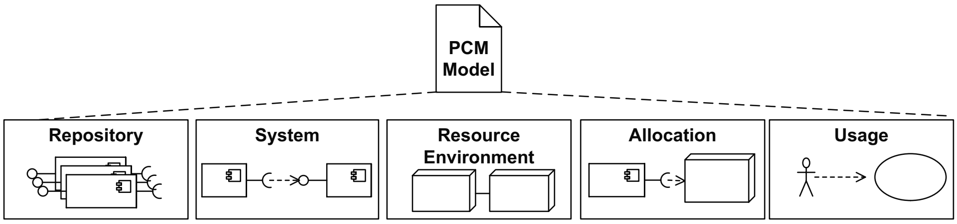


Figure 2: Palladio Model with all artifacts.

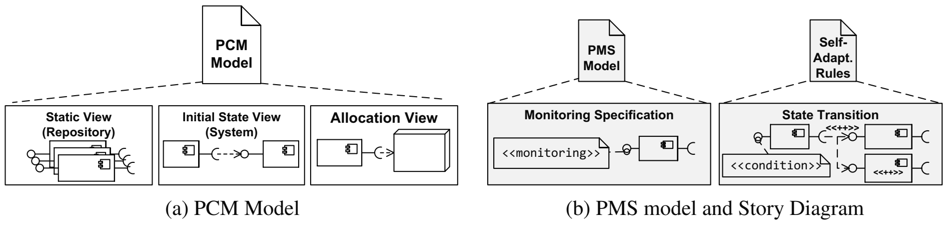


Figure 3: Modeling views: (a) PCM mapped to proposed views and (b) newly implemented views.

mance prototypes. We use ProtoCom to generate a performance prototype. Subsequently, we extend the generated prototype with self-adaptive functionality, in order to validate the correctness of the performance predictions of our SimuLizar approach.

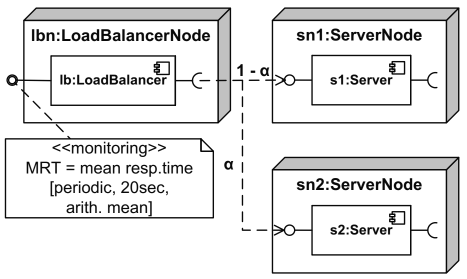
4 SimuLizar

SimuLizar extends Palladio in two areas: (a) the modeling and (b) design-time performance analysis of self-adaptive systems.

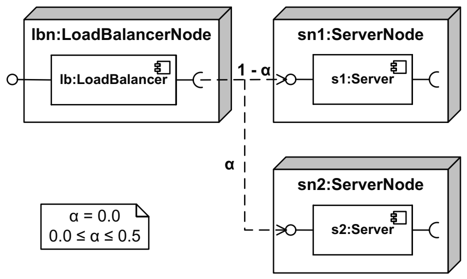
Modeling Approach. SimuLizar provides a modeling approach for self-adaptive systems based on ideas presented in [Bec11]. In SimuLizar, a self-adaptive system model consists of two viewpoints with several views each. First, a system type viewpoint consisting of three views: a *static view*, a *monitoring specification view*, and an *allocation view*. Second, a runtime viewpoint including the *initial state view* and the *state transition view*.

Where appropriate, we have mapped the required views to existing artifacts of PCM, as illustrated in Figure 3a. The static view is covered by the PCM repository, the initial state view can be mapped to the PCM system view, and the allocation is covered by PCM's allocation view.

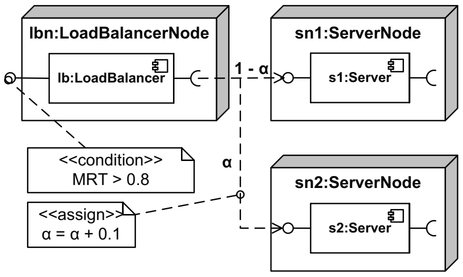
Until now, PCM did not offer modeling views for the monitoring view or the state transition view. To fill these gaps, we introduce two new artifacts in SimuLizar, as illustrated in Figure 3b. First, the Palladio Measurement Specification (PMS), a domain-specific language for the monitoring specification view. Second, self-adaptation rules for specifying the state transition view.



(a) system type viewpoint



(b) initial state view



(c) state transition view

Figure 4: (a) System Type View, (b) Initial State View, and (c) Transition state view.

With PMS, sensors for a self-adaptive system can be specified. A monitor specification consists of four characteristics: (1) a sensor location, (2) a performance metric type, (3) a time interval type, and (4) a statistical characterization. The sensor location specifies the place of the sensor within the system, for example a service call. Currently, we support the following performance metric types: waiting time, response time, utilization, arrival rate, and throughput. Each of these metrics can be measured at the specified sensor location in one of the three time interval types: in periodic time frames with the length Δt beginning at time 0.0, periodic time frames with length Δt and a delay of Δd from the start time, or within a single fixed time frame with start time t_{start} and end time t_{stop}. Finally, for each sensor a statistical characterization can be specified to aggregate the monitored data. We support the modes *none*, *median*, *arithmetic mean*, *geometric mean*, and *harmonic mean*.

Self-adaptation rules consist of a condition and a self-adaptation action. A condition has to reference a sensor and to provide a boolean term. If the boolean term evaluates to true, the self-adaptation action is triggered. The self-adaptation action part references elements in the PCM model. Variables of these elements can be set using the ≪*assign*≫ keyword, or new instances of model elements can be instantiated using the ≪++≫ keyword. We use Story Diagrams [vDHP+12] to formalize the condition as well as the self-adaptation action as described above.

Figure 4 illustrates a model of our motivating example using our SimuLizar modeling approach. We model the system according to our initial design ideas in Section 2. The system type viewpoint, Figure 4a, uses the PCM allocation view annotated with measurement

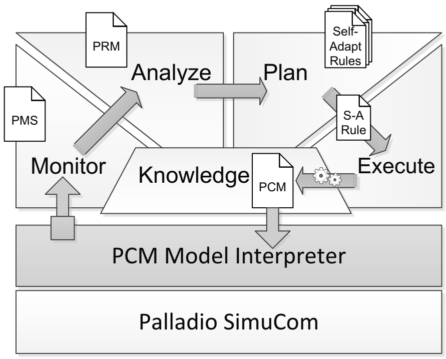


Figure 5: SimuLizar architecture.

specifications, which are defined via the PMS. In this example, the component LoadBalancer is deployed on a node lbn. The LoadBalancer component is connected to two Server components via its required interface. The two server nodes are deployed on two different ServerNodes sn_1 and sn_2. The monitoring annotation connected to the provided interface of the LoadBalancer component specifies a monitor "MRT" for the mean response time (arithmetic mean) in periodic time frames with the length of $\Delta t = 20\ seconds$. On one hand, this should reduce the monitoring overhead and prevent the system to reconfigure every time a response exceeds the target response-time of 0.8 seconds. On the other hand, it is necessary to detect whether R1 is fulfilled. The initial state view, Figure 4b, illustrates the initial configuration of the system. The variable α denoting a branch probability is set to 0.0, meaning that the whole workload is initially handled by server node sn_1 (R2). In the state transition view, Figure 4c, the variable α is set to $\alpha + 0.1$ if the condition MRT > 0.8 holds. This reflects the system's reconfiguration to satisfy R3.

Simulation Tool. Self-adaptive systems modeled with SimuLizar can be simulated using SimuLizar's MDSPE tool for self-adaptive systems, e.g., in order to get response time predictions of the overall system. The building blocks of this tool can be mapped to the MAPE-K feedback loop, as illustrated in Figure 5. In SimuLizar the managed element is the modeled simulated self-adaptive system. The simulated system is monitored, the monitoring results are analyzed, reconfiguration is planned, and a reconfiguration is executed on the simulated system if required. Reconfigurations are model-transformations of this PCM model.

A *PCM model interpreter* traverses the PCM model for each single user request, as specified in the PCM usage view. It utilizes the simulation engine, *SimuCom*, for simulating the user and system behavior including the simulation of resources. During the simulation, the simulated system is *monitored* and measurements are taken as specified via *PMS*. Whenever the PCM model interpreter arrives at a monitor place, it simulates a measurement as specified in the PMS model. Once new measurements are available the runtime model of the system, the Palladio Runtime Measurement Model (*PRM*), is updated with the newly taken measurements. An update of the PRM, i.e., new measurements, triggers the *planning* phase.

In the planning phase all *self-adaptation rules* are checked. If the condition of a rule holds, the corresponding self-adaptation action is *executed*, i.e., the PCM model is transformed.

During the execution phase, the translated model-transformation is applied to the PCM model. Subsequently, the interpreter uses the transformed PCM model / reconfigured system for new users.

5 Evaluation

To evaluate the applicability of SimuLizar we model the self-adaptive load balancer, as presented in Section 4, and analyze its performance. First, we explain how we conducted the evaluation. Second, we present the results of SimuLizar's predictions. We discuss the quality of SimuLizar's predictions compared to the measurements taken at the performance prototype. Subsequently, we discuss the possibilities to reason about transient states of self-adaptive systems. Finally, we point out the limitations of our SimuLizar approach.

Conducting the Evaluation. We evaluate SimuLizar according to four criteria. First, we validate whether our modeling approach for self-adaptive systems is sufficient to model the important aspects of self-adaptive systems (**C1**). Our second criterion is whether SimuLizar allows us to predict the performance of a self-adaptive system in the transient phases (**C2**). Third, we evaluate whether SimuLizar's predictions do not significantly deviate from the performance of a self-adaptive system performance prototype, i.e., it can be used to evaluate design alternatives (**C3**). The fourth criterion for our evaluation is whether SimuLizar's performance prediction helps to identify unsatisfactory self-adaptation logic which traditional MDSPE did not aim at (**C4**).

In order to evaluate C1, we model the load balancer system[1] using our SimuLizar modeling approach as described in the previous section. Next, we simulate the load balancer and predict its performance using SimuLizar's MDSPE tool to evaluate C2. Specifically, we simulate the system to initially start in a high load situation in order to trigger the system's self-adaptation. We expect the system to self-adapt immediately after it detects the high load situation until the response times decrease.

In order to evaluate C3, we compare SimuLizar's performance prediction to the performance predictions from a generated and manually extended performance prototype. For this, we generate the performance prototype from the load balancer model using Palladio's ProtoCom tool. We extend this prototype with a MAPE-K feedback loop to add self-adaptive behavior.

We analyze the response times for both the system model simulated with SimuLizar's MDSPE tool as well as the ProtoCom performance prototype in the high load situation. Both, SimuLizar's MDSPE tool and ProtoCom, are configured to take 1000 measurements, i.e., simulate/measure 1000 user requests. We calibrated our prototype system's resource demands according to the model, i.e., a user request has a CPU demand of 0.3 seconds in

[1]The full model can be obtained via http://goo.gl/O480s (anonymous user access)

the prototype system as well. We expect that the self-adaptive system is able to recover from the high load situation within the time of 1000 measurements.

To evaluate C4, we compare the performance predictions of SimuLizar to a series of steady state analyzes. For this, we designed a queuing network and simulate it with JSIMgraph[2]. Our expectation is, that our modeled load balancer system eventually self-adapts itself such that its configuration is in a state in which a steady state analysis implies that R1, response time lower than 0.8 seconds, is fulfilled. We configure JSIMgraph to calculate response times and utilizations for ServerNodes sn_1 and sn_2; we set the confidence interval to 99% with a maximal relative error of 3%.

Results. We first present SimuLizar's performance prediction results and the measurements taken with the performance prototype. Second, we present JSimgraph's steady state analysis results for all possible configurations of the evaluation system.

Figure 6 shows the (interpolated) time series of response times of our specified evaluation usage scenario. The time series show that initially all requests are answered by application server sn_1. Both SimuLizar and the performance prototype measurements, show steadily increasing response times. Hence, application server sn_1 is in a high load situation. Approximately after 20 seconds the first requests are answered by application server sn_2 in both series. This indicates that the load balancer has triggered a self-adaptation, which is further confirmed by plateaus in the increase of the response times from application server sn_1. In both time series, we can finally observe that the system triggers self-adaptation five times. Only after the several self-adaptation, approximately after 80 seconds, the responses times of both server ServerNode sn_1 and ServerNode sn_2 have similar values. Any deviations of the predicted and measured response times occur due to the random load generation and balancing strategy of our evaluation system.

We conclude from these results that self-adaptive systems can be modeled (C1) and their performance within transient phases can be predicted (C2). Furthermore, both time series do not deviate significantly, i.e., less than 30%, from each other. In the performance engineering community such a deviation is considered sufficient to differentiate between design alternatives (C3).

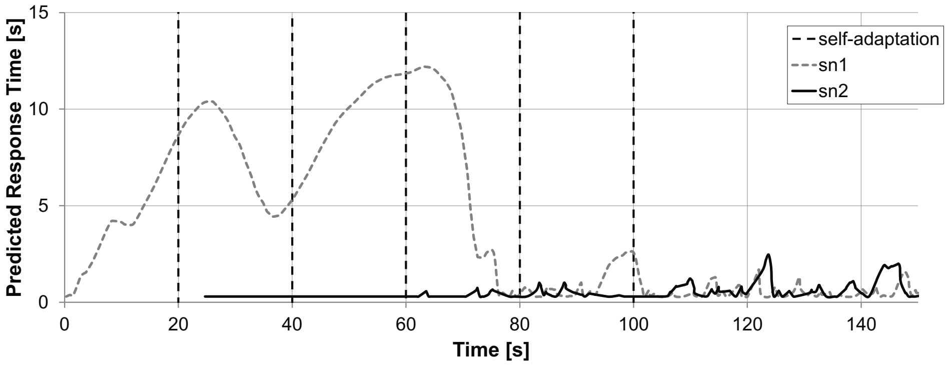
To evaluate C4, Figure 7a shows the queuing network we have simulated with JSIMgraph. The source spawns new users with the same rate as the load balancer example model. It routes users with the probability of $1 - \alpha$ to server sn_1 and with probability α to server sn_2. Server sn_1 and Server sn_2 are both queues with unlimited queue size and a constant service time of 0.3s.

The results for mean response time and mean utilizations of the JSIMgraph simulation are denoted in Figure 7b. Each row represents one configuration of the system, i.e., different values for α. When $\alpha = 0.0$ the mean response time is 4.421 seconds. The utilization of server sn_1 is 0.9299, i.e., 93%, which indicates a high load situation. The utilization of server sn_2 is 0%, because no users are routed to it.

Interestingly, the steady state analysis shows, that the mean response time with $\alpha = 0.3$ is 0.759 seconds, which falls below our threshold of a mean response time of 0.8 seconds.

[2]JSIMGraph is included in the Java Modeling Tools which can be obtained via http://jmt.sourceforge.net/

(a) SimuLizar Predictions



(b) Performance Prototype Measurements

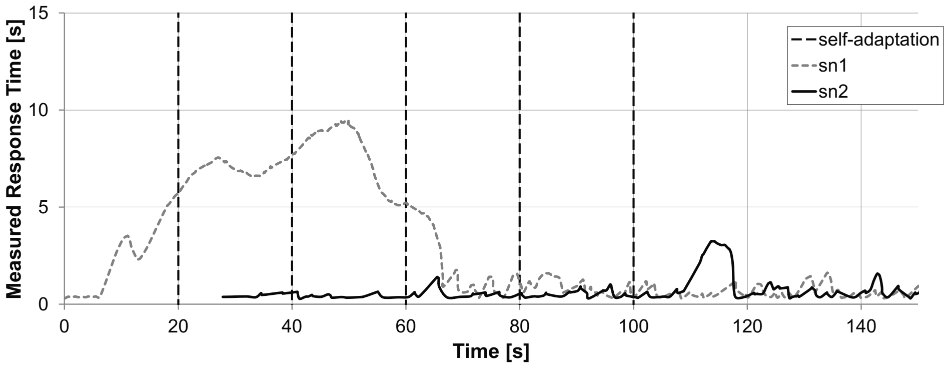
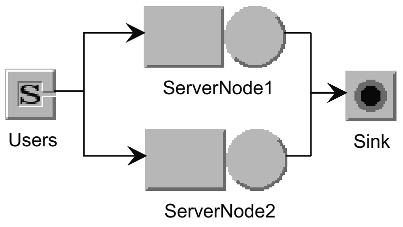


Figure 6: The (interpolated) time series for the response times of (a) SimuLizar's simulated system and (b) measured response times for the performance prototype. The vertical axes represent response times of a single user request. The horizontal axes represent the execution/simulation time. Each curve represents the response times of a series of single user request. The dashed grey curve represent response times for user request delegated to application server sn_1; the solid black curve represent response times for user requests delegated to application server sn_2.



(a) queuing network

α	**Mean RT**	**Mean Utilization**	
	System	Server 1	Server 2
0.0	4.421	0.9299	0.0000
0.1	1.743	0.8363	0.0932
0.2	1.042	0.7513	0.1858
0.3	0.759	0.6537	0.2786
0.4	0.628	0.5563	0.3600
0.5	0.580	0.4677	0.4664

(b) response times and utilizations

Figure 7: Evaluation system represented as (a) queuing network and (b) steady state analysis for each state.

Nevertheless, we can observe that our evaluation system triggers two more self-adaptations until $\alpha = 0.5$. From this, we can conclude that the effects of setting α to 0.3 do not immediately effect the mean response time to fall below the threshold. Even in the next mean response time batch the threshold is exceeded and another self-adaptation is triggered.

As these queuing network shows, in this scenario adapting α to at least 0.3 right from the start would help to recover the system from the overload situation faster. This means that our adaptation rules violate R3 because the system does not recover as fast as possible. Hence, we identified an unsatisfactory self-adaptation logic (C4).

Limitations. SimuLizar as well as our evaluation still underlie several assumptions and limitations. First, SimuLizar cannot be considered as sufficient for a comprehensive performance prediction of self-adaptive systems yet. Second, our evaluation is limited due to some threats to validity.

Self-adaptive behavior is usually triggered to cope with environmental changes, e.g., changing customer arrival rates. However, the PCM usage view we are using here is static, i.e., a dynamically or randomly changing environmental cannot be modeled. Thus, we are forced to model usage scenarios in which the condition of a self-adaptation rule holds from the start. This limits the scope of a simulation to only a set of rules which are triggered for the modeled static usage scenario.

Due to the fact that SimuLizar simulates self-adaptive systems, it underlies some assumptions with respect to the self-adaptation of a system. In SimuLizar and self-adaptive system models we neglect resource consumption of self-adaptations. Furthermore, we do not handle exceptions that might occur during the self-adaptation.

Our evaluation is mainly limited due to the implementation of our performance prototype. First, the measurements taken from the performance prototype rely on a prior calibration of the machine it runs on. Although this calibration has been tested and evaluated before, the actual generated load is subject to minor deviations, e.g., due to system background load. This leads to inaccurate measurement results and may bias our evaluation. Second, since our performance prototype is a distributed system, asynchronous clocks are another threat to the validity of the taken measurements and our evaluation. Third, the usage we specified for our evaluation example contains an exponentially distributed arrival rate of the customers. Hence, the arrivals are random and are consequently not generalizable, i.e., the actual inter-arrival rate may vary in each evaluation run for both, the simulation and the performance prototype.

6 Related Work

In recent years, several related articles about self-adaptive system modeling and model-driven performance engineering have been published. We have surveyed them in [BLB12].

In [FS09], Fleurey and Solberg propose a domain-specific modeling language and simulation tool for self-adaptive systems. With the provided modeling language self-adaptive systems can be specified in terms of variants using an EMF-based meta-model. Furthermore, functional properties of the specified system can be checked and simulations can be performed. SimuLizar's focus is on the performance aspect of self-adaptation in contrast to the approach by Fleurey and Solberg which focuses on functional aspects.

The D-KLAPER approach [GMR09] is an MDSPE approach which can be applied to self-adaptive systems. D-KLAPER does not aim at providing modeling support for software design models. Instead software design models annotated with performance-relevant resource demands have to be transformed into D-KLAPER's intermediate language. D-KLAPER then provides the necessary tools to analyze a self-adaptive system specification provided in the intermediate language. However, in contrast to our focus on transient phase analysis, systems analyzed with D-KLAPER are considered to be in a steady state when analyzed.

Huber et al. [HvHK$^+$12] present a modeling-language for specifying self-adaptive behavior in the context of software design models. Their work is integrated within the Descartes project, which envisions self-adaptive cloud systems enhanced with runtime performance analysis. The focus of the Descartes project is on runtime adaptation in contrast to our aims of design-time performance analysis of self-adaptation.

QoSMOS [CGK$^+$11] is an approach to model and implement self-adaptive systems whose self-adaptation is driven by performance requirements. The system designer has to manually derive an analysis model from the architectural model. Manually derived analysis model serve as an initial input for the online performance analysis and its parameters are updated at run-time. In contrast, we provide a full tool chain for design-time performance analysis of self-adaptive systems that also able to adapt their structure.

7 Conclusion

In this paper, we presented SimuLizar, a model-driven software performance prediction approach for self-adaptive systems. We implemented our previously introduced modeling approach and provide a simulation tool for performance prediction.

Our evaluation shows that SimuLizar is applicable to model self-adaptive systems and predict their performance. Furthermore, SimuLizar enables the analysis of transient phases during a system's self-adaptation, and allows to identify unsatisfactory self-adaptation logic.

We are working towards further enhancements of our approach. First, we plan to provide a domain-specific modeling language for specifying dynamic system workloads, i.e., randomly changing loads of simulated users. Second, we are currently implementing automatic generation of a performance prototype with self-adaptation capabilities from models specified with SimuLizar. Finally, we plan to enhance our tool to support developers modeling and evaluating self-adaptive systems, i.e., automatically detecting self-adaptation design flaws and providing design alternatives.

References

[BDH08] S. Becker, T. Dencker, and J. Happe. Model-Driven Generation of Performance Prototypes. In S. Kounev, I. Gorton, and K. Sachs, editors, *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *LNCS*, pages 79–98. Springer, 2008.

[Bec11] S. Becker. Towards System Viewpoints to Specify Adaptation Models at Runtime. In *Proceedings of the Software Engineering Conference, Young Researches Track (SE 2011)*, volume 31 of *Softwaretechnik-Trends*, 2011.

[BKR09] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[BLB12] M. Becker, M. Luckey, and S. Becker. Model-Driven Performance Engineering of Self-Adaptive Systems: A Survey. In *Proc. of the 9th ACM SigSoft Int. Conf. on Quality of Software Architectures*, June 2012.

[CDI11] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[CDMI06] V. Cortellessa, A. Di Marco, and P. Inverardi. Software performance model-driven architecture. In *Proc. of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1218–1223, 2006.

[CGK$^+$11] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. on Software Engineering*, 37:387–409, 2011.

[FS09] F. Fleurey and A. Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *Model Driven Engineering Languages and Systems*, volume 5795 of *LNCS*, pages 606–621. Springer, 2009.

[GMR09] V. Grassi, R. Mirandola, and E. Randazzo. Model-Driven Assessment of QoS-Aware Self-Adaptation. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 201–222. Springer, 2009.

[HvHK$^+$12] N. Huber, A. vn Hoorn, A. Koziolek, F. Brosig, and S. Kounev. S/T/A: Meta-Modeling Run-Time Adaptation in Component-Based System Architectures. In *9th IEEE Int. Conf. on e-Business Engineering (ICEBE 2012)*, September 9-11 2012.

[Mur04] R. Murch. *Autonomic Computing*. IBM Press, 2004.

[vDHP$^+$12] M. von Detten, C. Heinzemann, M. C. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt. Story Diagrams Syntax and Semantics. Technical Report tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, July 2012. Ver. 0.2.

Taking the Pick out of the Bunch –
Type-Safe Shrinking of Metamodels

Alexander Bergmayr[1], Manuel Wimmer[2], Werner Retschitzegger[3], Uwe Zdun[4]

[1]Vienna University of Technology, Austria
[2]Universidad de Málaga, Spain
[3]Johannes Kepler University Linz, Austria
[4]University of Vienna, Austria

[1]bergmayr@big.tuwien.ac.at, [2]mw@lcc.uma.es
[3]retschitzegger@cis.jku.at, [4]uwe.zdun@univie.ac.at

Abstract: To focus only on those parts of a metamodel that are of interest for a specific task requires techniques to generate metamodel snippets. Current techniques generate strictly structure-preserving snippets, only, although restructuring would facilitate to generate less complex snippets. Therefore, we propose metamodel shrinking to enable type-safe restructuring of snippets that are generated from base metamodels. Our approach allows to shrink a selected set of metamodel elements by automatic reductions that guarantee type-safe results by design. Based on experiments with 12 different metamodels from various application domains, we demonstrate the benefits of metamodel shrinking supported by our prototypical implementation build on top of the Eclipse Modeling Framework (EMF).

1 Introduction

With the adoption of Model-Driven Engineering (MDE), more and more modeling languages are defined based on metamodels. Large metamodels such as the current UML metamodel rely typically on complex structures which are challenging to grasp. For instance, manually identifying the effective classifiers and features of a certain diagram type in the metamodel requires much effort. The UML classifier `Class` transitively inherits from 13 other classifiers and provides 52 structural features which shows that even putting the focus only on one classifier can already be challenging. Allowing one to snip out a subset of a metamodel would relieve one from the full complexity imposed by the base metamodel. For instance, this would be beneficial for automating model management tasks by formulating model transformations on metamodel subsets instead of their base metamodels [SMM$^+$12].
However, if the extraction of an effective metamodel subset is aspired, we are not only confronted with the selection of classifiers and features of the base metamodel, but also with their reduction to actually shrink the number of classifiers or generalizations. Such reductions can be useful because the design structure of the base metamodel may not be

necessarily a good choice for the extracted metamodel subset. It has to be noted that a naive reduction of classifiers may lead to inconsistencies such as (i) broken inheritance hierarchies, (ii) missing feature containers, and (iii) dangling feature end points which require special attention in the shrinking process.
In this work, we propose an approach to automatically shrink metamodels. The result of shrinking a metamodel is what we call a metamodel snippet. A metamodel snippet is considered as a set of metamodel elements, i.e., classifiers and features, originally defined in the base metamodel. We provide refactorings to restructure initially extracted metamodel snippets. Thereby, we enable the reduction of metamodel elements that may become obsolete by a restructuring and enhance the understandability of metamodel snippets. For instance, consider the reductions of deep inheritance hierarchies that may not be necessarily required for a metamodel snippet. Applying reductions to metamodel snippets distinguishes our approach from some recent work [SMBJ09, KMG11, BCBB11] that generate strictly structure-preserving results. Reductions enable metamodel snippets with a lower number of classifiers and features, and flatter inheritance hierarchies. Our proposed reductions are type-safe in the sense that extensional equivalence[1] between extracted and reduced metamodel snippets is guaranteed by design. Our approach relies on 4 operators: (i) *Select* to define the initial set of classifiers and features, (ii) *Extract* to generate a structure-preserving metamodel snippet, (iii) *Reduce* to shrink the metamodel snippet, and (iv) *Package* to compose all elements into a metamodel snippet that can be used as any other metamodel.
The structure of this paper is as follows. In Section 2, we introduce our metamodel shrinking approach. A prototype has been implemented based on the Eclipse Modeling Framework[2] (EMF) that is presented in Section 3. We critically discuss the results of applying our prototype for 12 metamodels in Section 4. A comparison of our approach to related work is presented in Section 5, and finally, lessons learned and conclusions are given in Section 6.

2 Metamodel Shrinking

Our proposed metamodel shrinking approach relies on OMG's MOF[3] abstraction level and is therefore metamodel agnostic. A metamodel snippet $MM_{snippet}$ is produced by applying our approach to a base metamodel MM_{base} as shown in Fig. 1.

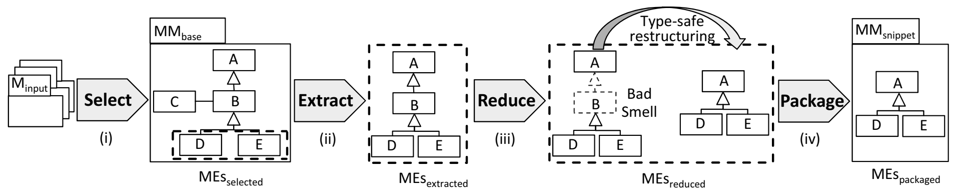


Figure 1: Overview of metamodel shrinking approach

[1]A metamodel defines a collection of models, i.e., extensions, that conform to it [VG12].
[2]`http://www.eclipse.org/modeling/emf`
[3]`http://www.omg.org/mof`

We propose a 4-step metamodel shrinking process. Each step is accompanied by a dedicated operator. The *Select* operator identifies based on a set of models M_{input} all metamodel elements MEs, i.e., classifiers and features, required to produce these models. However, a selection of metamodel elements driven by collecting only directly instantiated classifiers may not be sufficient. Indirectly instantiated classifiers, and thus, the classifier taxonomy need to be additionally considered to end up with a valid $MM_{snippet}$. This is exactly the task of the *Extract* operator. The operator produces a set of connected metamodel elements that strictly preserves the structure of the base metamodel. Subsequently, the *Reduce* operator shrinks the result of the extraction step. To achieve a reduction of metamodel elements, we apply well-known refactorings [Opd92, HVW11] to the extracted $MM_{snippet}$. In this way, deep inheritance hierarchies without distinct subclasses are reduced. Indicators for refactorings are often referred to as 'bad smells'. For instance, in Fig. 1, we assume that class `B` 'smells bad', because it does not contain any feature for its subclasses. By removing the class and linking its subclasses directly to its superclass, the smell is eliminated. Finally, the *Package* operator serializes the reduced set of metamodel elements into a persistent metamodel. In the following subsections, we discuss these 4 steps in more detail.

2.1 Selection of required metamodel elements

In the selection step, all classifiers and features of interest are determined. This *explicit* set of metamodel elements shall be by all means part of the metamodel snippet. We support the selection step by allowing models as input for the *Select* operator. They serve as a basis to automatically identify the required metamodel elements to represent them. A potential model for selecting metamodel elements is sketched in Fig. 2.

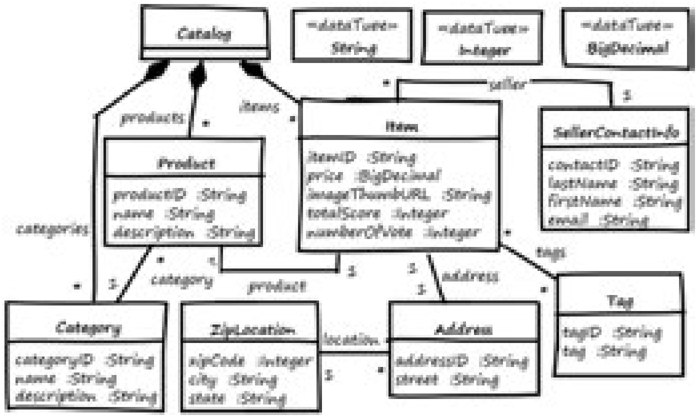


Figure 2: Class diagram of 'PetStore Navigability' application scenario

The UML class diagram shows an excerpt of the 'PetStore' scenario as introduced by Sun. The idea is to create a metamodel snippet of the UML metamodel that is effectively required to express the ‘PetStore’ class diagram. We use this scenario throughout the remaining sections as a running example.

2.2 Extraction of selected metamodel elements

Since the explicitly selected set of metamodel elements may not be sufficient, implicit metamodel elements that glue them together need to be additionally identified. We call these elements *implicit*, as they are computed from the *explicit* set of metamodel elements produced by the *Select* operator. The *Extract* operator traverses the base metamodel and produces an enhanced set of metamodel elements by addressing (i) explicitly selected metamodel elements, (ii) the inheritance closure of explicitly selected classes, (iii) classes that serve as container of explicitly selected inherited features, and (iv) features contained by implicitly selected classes. As a result, an initial metamodel snippet is produced.
Considering the excerpt of our example in Fig. 3, `Class` was explicitly selected as the ‘PetStore’ class diagram contains `Class` instances. For instance, `Encapsulated-Classifier` and `StructuredClassifier` were implicitly added in addition to the explicit selection as they are in the inheritance closure of `Class`. They are considered as a means to provide a connected set of metamodel elements decoupled from the base metamodel. The decoupling is achieved by removing features of classes which reference classes not contained in the set of selected metamodel elements. In our example, 32 features were removed in the extraction step. In the reduction step, implicitly selected metamodel elements are potential candidates for becoming removed again.

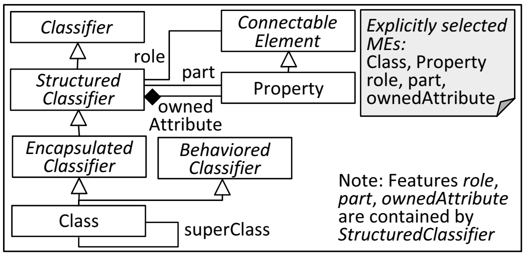


Figure 3: Extracted metamodel elements of our example

2.3 Reduction of extracted metamodel elements

This step aims to shrink the initial metamodel snippet. Manually identifying useful reductions is cumbersome when the number of involved metamodel elements is overwhelming and interdependencies between these reductions need to be considered. For instance, in our example, 101 metamodel elements were extracted from which 34 were reduced by applying 27 refactorings as a means to achieve a type-safe restructuring. The *Reduce* operator indicates extracted metamodel elements for reduction according to a given reduction configuration RC. Such a configuration can be adapted to control the result of the *Reduce* operator. We introduce two concrete reduction configurations depicted in Fig. 4.

		RC	
	Extracted metamodel element	**exact**	**extensive**
Classifier	Explicit concrete `Class`	k	k
	Implicit concrete `Class`	k	k
	Explicit abstract `Class`	k	k
	Implicit abstract `Class`	c^1	r
	Explicit/implicit `Datatype`	k	k
	Explicit/implicit `Enumeration`	k	k
Feature	Explicit `Feature`	k	k
	Implicit `Feature`	c^2	c^3

Legend for table cells:
RC ... Reduction Configuration
k...keep
r...reduce
c...conditional reduce

OCL expressions for conditions:
fSet is assumed to be the collection of all selected features
c^1: **context** Class **def if** fSet->exists(f|self.ownedAttribute ->includes(f)) **then** 'k' **else** 'r' **endif**
c^2: **context** Feature **def if** self.lower>=1 or self.isDerived=true **then** 'k' **else** 'r' **endif**
c^3: **context** Feature **def if** self.lower>=1 **then** 'k' **else** 'r' **endif**

OCL expressions for metamodel elements:
Concrete Class: : **context** Class **inv** not self.isAbstract
Abstract Class: : **context** Class **inv** self.isAbstract

Figure 4: Exact and extensive reduction configuration (RC)

The reduction of deep inheritance hierarchies in a metamodel snippet is the rationale behind the *exact* configuration. Implicitly extracted classifiers in the shape of abstract classes, e.g., `EncapsulatedClassifier` or `StructuredClassifier` in our example, are indicated for reduction. Considering the UML metamodel, `Class` originally inherits from `EncapsulatedClassifier` which inherits from `StructuredClassifier` that in turn inherits from `Classifier`. They are all well justified in the context of the base metamodel, but may not be as important for metamodel snippets. In our example, the context was narrowed to UML's data modeling capabilities. As a result, `Encapsulated-Classifier` is indicated for reduction when applying the exact reduction configuration as shown in Fig. 5.

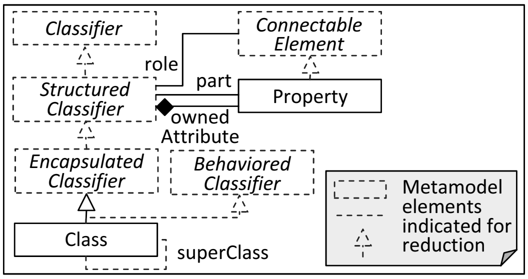


Figure 5: Metamodel elements indicated for reduction with extensive RC of our example

Similar to reducing implicitly selected classifiers, features with these characteristics are candidates for reduction except they are defined as being `required` or `derived`. While the rationale for the former exception is obvious, derived features are kept to avoid loss of information due to their calculated instead of user-defined value. The `superClass` feature of `Class` is an example in this respect.
In contrast to the exact reduction configuration, the *extensive* reduction configuration in-

dicates `derived` features for reduction. Since we did not apply UML's generalization concept for classes in our example, the `superClass` feature was reduced by the extensive reduction configuration. Rather than keeping implicitly selected abstract classes that serve as feature containers, in the extensive reduction configuration the intension is to reduce them without exceptions. Both `EncapsulatedClassifier` and `StructuredClassifier` are, thus, indicated for reduction in our example.
However, indicating metamodel elements for reduction is only half the way to obtain a useful metamodel snippet since naively reducing classes may lead to inconsistencies. We encountered three possible inconsistencies in our approach: (i) broken inheritance hierarchies, (ii) missing feature containers, and (iii) dangling feature end points. In our example, the generalization relationship of `Class` needs to be relocated, and the feature `role` requires a new container and a new type when the indicated classes are actually reduced. To overcome these unintended effects, we conduct a type-safe restructuring by relying on well-known object-oriented refactorings [Opd92] adapted to the area of (meta)modeling [HVW11]. In Fig. 6, we introduce refactorings for the restructuring of metamodel snippets.

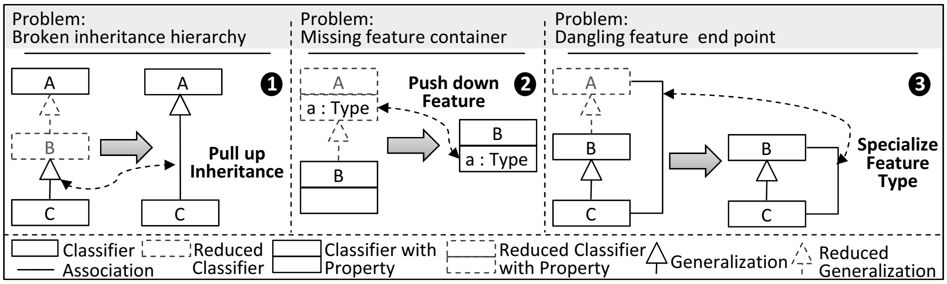


Figure 6: Refactoring techniques for type-safe metamodel restructuring

They achieve (i) relocating generalization relationships by pulling up the relationship ends to super-superclasses, (ii) moving features if their base containers were reduced by pushing down features from superclasses to subclasses and (iii) reconnecting dangling feature end points by specializing feature types from superclasses to subclasses. These refactorings are, by design, type-safe since they operate on the inheritance hierarchies imposed by the base metamodel and respect type substitutability [GCD$^+$12]. Refactorings are considered as events triggered by the need to usefully conduct indicated reductions on the metamodel snippet.
Pull up inheritance. This refactoring enables relocating generalization relationships. A relationship end that would be a dangling reference as a result of reducing classes which lie in between of other classes in the inheritance hierarchy has to be relocated. Such a gap in the inheritance hierarchy is closed by pulling up the relationship end to the least specific superclasses of the reduced class.
In our example, the generalization relationship of `Class` needs to be relocated as `Class` indirectly specializes `Classifier` and both classes are kept after the reduction. As a result, `Class` inherits from `Classifier` serving as the replacement for the more specific classes `EncapsulatedClassifier` and `StructuredClassifier` as shown in Fig. 7. The indicated reduction for `Classifier` is relaxed because several subclasses

such as `Association` or `Datatype` inherit features contained by `Classifier`. A reduction of `Classifier` would lead to duplicated features in the corresponding subclasses. We decided to prevent such an effect as from an object-oriented design perspective this is not desirable.

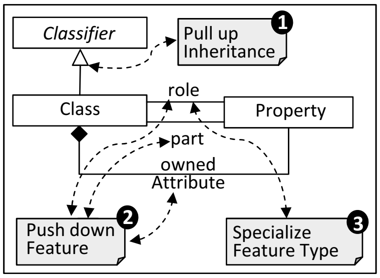


Figure 7: Refactored metamodel elements of our example

Push down feature. This refactoring supports moving features from one to another container by going down the inheritance hierarchy. Features for which a new container is required are moved down to the most generic subclass. This could lead to reverting back to a previously reduced container to avoid duplicated features (cf., `Classifier`). Reduced containers become in such a situation reintroduced.
In our example, the features `part`, `ownedAttribute` and `role` are moved to a container compatible with `StructuredClassifier` since this class was reduced.
Specialize feature type. This refactoring addresses reconnecting dangling references of associations or compositions between classes. Similar to the push down feature refactoring, the most generic subclass is selected for the type specialization.
In our example, the feature `role` needs to be reconnected to a type compatible with `ConnectableElement`. As a result, the type of feature `role` is changed from `ConnectableElement` to `Property`. Searching for the most generic subclass may lead to a similar situation like for the push down feature refactoring, i.e., the reintroduction of previously reduced classes.

2.4 Packaging of metamodel snippet

To enable dedicated modeling tools to work with metamodel snippets, the metamodel snippets need to be materialized. The *Package* operator takes the result of the *Reduce* operator and reconciles the shrinked set of metamodel elements into a serialized $MM_{snippet}$.
In Fig. 8, the complete result for our example is shown, i.e., the part of the UML metamodel required to express the ‘PetStore’ model. We applied the extensive reduction configuration which resulted in 22 classifiers and 45 features. Overall, 8 different classes were actually instantiated as indicated by the dashed framed classifiers in Fig. 8. Using only this set of classes would require to inject the same features multiple times in different classes which would lead to a metamodel snippet with poor design quality. Thus, by applying the

proposed refactorings, we can find an effective trade-off between a carefully selected set of classes and design quality.

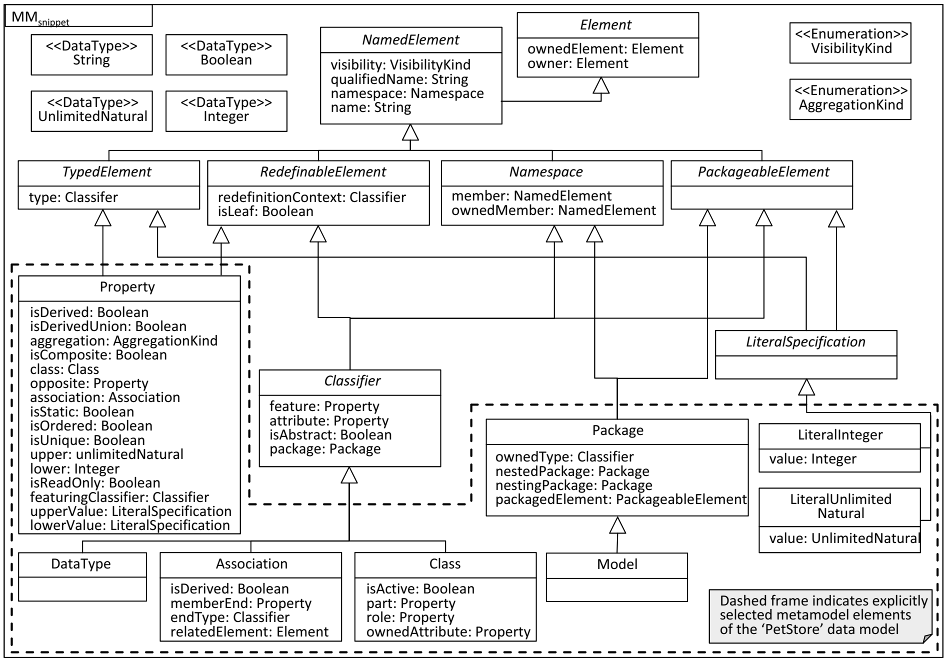


Figure 8: Metamodel snippet of our example

3 Prototypical Implementation: EMF Shrink

To show the feasibility of the metamodel shrinking approach, we implemented a prototype based on EMF. To operationalize our proposed operators, we implemented them based on a pipeline architecture. While the *Select* and *Extract* operator have been straightforwardly implemented on the basis of EMF, the realization of the *Reduce* operator required more care because potentially occurring side effects as a result of applied metamodel refactorings needed to be handled. An example in this respect is the reintroduction of previously reduced classes because they may have effects on the inheritance hierarchies. For that reason, we heavily exploited EMF's change notification mechanism to trigger precalculated relaxations on refactorings that become obsolete as metamodel shrinking progresses. The *Package* operator generates independently of the position in the pipeline valid metamodel snippets conforming to Ecore, i.e., EMF's meta-metamodel. This was helpful for validating and interpreting the results of our operators. We used an automatic validation by executing well-formedness constraints and manual validation by inspecting the generated snippets in the graphical modeling editor for Ecore models. Implementation code for metamodel

snippets can be generated by applying EMF's generation facility.
Customizations in a metamodel's implementation code that also relate to a metamodel snippet requires special consideration. In our running example, the value of the feature `ownedElement` in `Element` is a derived value. For that reason, we additionally realized, based on EMF's adapter concept, generic adapter factories that allow the integration of customized implementation code into generated implementation code of a metamodel snippet as far as model manipulation operations are concerned. Whenever model elements are created with a metamodel snippet, in the background corresponding model elements as instances of the base metamodel are created. As a result, model elements adapt each other in the sense of a delegation mechanism and are kept synchronized via change notifications. Further details regarding our implemented prototype can be found online[4].

4 Evaluation

To evaluate the applicability of our proposed approach, we performed experiments by shrinking 12 metamodels based on given models as summarized in Fig. 9.

			Results in absolute numbers									
	Total classifiers	Total features	Extracted classifiers	Extracted features	Packaged classifiers		Packaged features		Applied refactorings	
Metamodel[#]					RC1[+]	RC2[+]	RC1[+]	RC2[+]	RC1[+]	RC2[+]
ACME[1]	19	26	7	4	5	5	4	4	2	2
Agate[1]	72	204	16	35	12	11	29	29	6	8
BMM[2]	47	66	13	22	11	11	16	16	4	4
BPMN2[3]	147	458	35	73	34	31	64	40	2	7
HTML[4]	62	112	8	16	7	7	14	14	2	2
iStar[5]	44	101	19	41	18	18	30	30	0	0
PNML[6]	42	80	14	16	10	10	16	16	2	2
Requirement[4]	50	53	22	23	22	20	20	20	0	8
SBVR[4]	332	361	20	18	20	20	11	11	0	0
SQLDDL[4]	20	27	13	21	13	13	20	20	0	0
SysML[7]	307	621	38	84	33	25	64	38	1	15
UML[4]	264	586	36	65	29	22	52	45	6	27
			Extract		Package				Reduce	

[#])From AtlanMod Metamodel Zoo

[+])RC1 ... Exact reduction configuration
[+])RC2 ... Extensive reduction configuration

Source of used models:
[1]Order Processing System (IBM developerworks)
[2]JK Enterprises (IBM developerworks)
[3]Hardware Retail Process (OMG BPMN Specification)
[4]PetStore use case (PetStoreNavigability)
[5]Toronto Civil Workers Strike (OpenOME Toronto)
[6]Vending Machine (Workflow Petri Net Designer WoPeD)
[7]Distiller (OMG SysML)

Figure 9: Quantitative experiment results in absolute numbers

The rationale behind our selection of *Metamodels* is mainly based on three criteria: (i) coverage of a wide range of applications domains (from business motivation and business process management over requirements engineering to software and systems engineering),

[4]`http://code.google.com/a/eclipselabs.org/p/emf-shrink`

(ii) consideration of small-sized to large-sized metamodels (from less than 50 to over 900 metamodel elements), (iii) involvement of metamodels with flat as well as deep inheritance hierarchies (from 2 up to 10 abstraction levels). *Total classifiers* and *Total features* refer to the size of a metamodel whereas *Extracted classifiers* and *Extracted features* represent the result of the extraction step in the respective experiments. Results from the *Package* operator w.r.t. the *Reduce* operator are presented for the two introduced reduction configurations. The less classifiers and features were packaged w.r.t. their corresponding number of extracted classifiers and features, the more metamodel elements were reduced. Finally, absolute numbers of applied refactorings are provided as a result of the *Reduce* operator. Typically, the more reductions of metamodel elements were achieved, the higher is the number of applied refactorings.
Based on the quantitative results, we critically discuss our approach from a qualitative perspective by investigating benefits and limitations of our approach. We consider 5 architectural metrics (cf., [BD02, MSJ04]) of packaged compared to extracted metamodel elements: (i) number of reduced classifiers, (ii) number of reduced features, (iii) mean features per classifier, (iv) mean inheritance hierarchy depth and (v) understandability.
As expected, benefits of metamodel shrinking take effect when large metamodels with deep inheritance hierarchies (cf., UML or SysML experiments) are considered. Considering Fig. 10, we could achieve to reduce in average $\approx 13\%$ of extracted classifiers with the exact reduction configuration, while an average value of $\approx 18\%$ could be achieved with the extensive one.

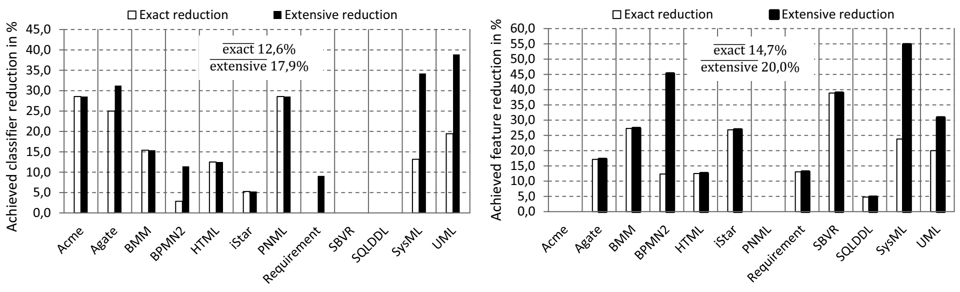


Figure 10: Achieved reductions of metamodel elements

Most classifier reductions could be achieved in the UML ($\approx 39\%$) and SysML ($\approx 34\%$) experiments as these metamodels cover many abstract classifiers for reasons of genericity or extensibility which is not necessarily required for metamodel snippets. Results of feature reductions w.r.t. the number of extracted features are in average in the range of $\approx 15\%$ to $\approx 20\%$. Reductions of features are generally easier to achieve as they lead typically not to inconsistencies in a metamodel snippet. However, in case of classifier reductions inconsistencies in a metamodel snippet may lead to reintroducing a previsouly reduced feature container to avoid duplicated features. Consequently, the rates of classifier reductions are lower than the rates of feature reductions particularly when the number of extracted features is much higher than the number of extracted classifiers.
Considering Fig. 11, the extensive reduction of classifiers may lead to an increase of mean features per classifier since features are pushed down from generic to more specific

classifiers (cf., Agate or UML experiments).

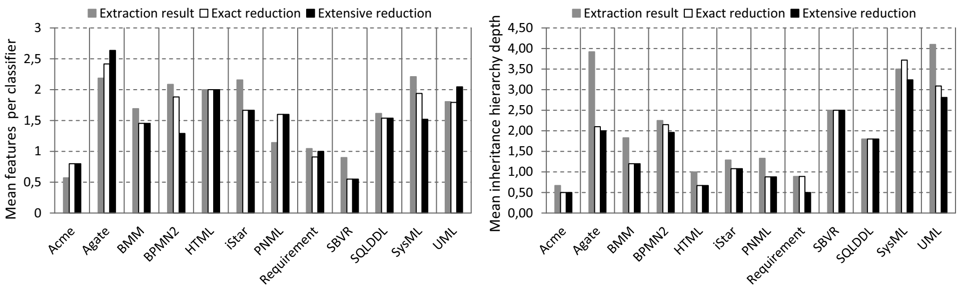


Figure 11: Mean features per classifier and mean inheritance hierarchy depth

Extensive reductions have generally positive effects on the mean inheritance hierarchy depth of metamodel snippets. The less classifiers are contained by metamodel snippets, the flatter inheritance hierarchies can be achieved while on the downside the less opportunities are available for features to be placed in appropriate classifiers. Generally, reductions of metamodel elements and potential refactorings lead inherently to structural differences between metamodel snippets and their base metamodels. Still, our metamodel shrinking approach generates metamodel snippets that are restructured in a type-safe way. Metamodel snippets enable expressing the models that were used to produce them in the same way as their base metamodels.
Finally, we applied the understandability metric of [BD02] to the metamodel snippets in our experiments as shown in Fig. 12.

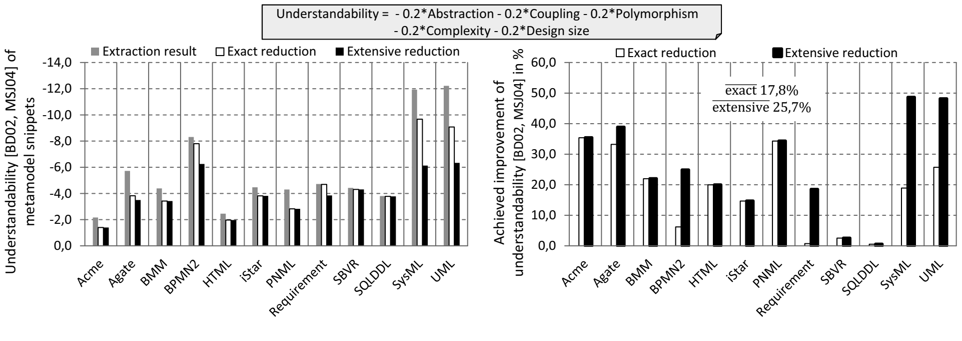


Figure 12: Understandability of metamodel snippets

We used a slightly adapted version of the originally proposed formula to calculate the understandability measures in two respects. First, we omitted the encapsulation property since private features are rarely used for metamodels, and second, we also omitted the cohesion property since our focus is on structural rather than behavioral features. As a result, our formula consists of 5 properties with equal weights, i.e., 0.2, that add up to 1 as suggested by [BD02]: (i) *Abstraction*, i.e., average number of ancestors for classifiers, (ii) *Coupling*, i.e., average number of features owned by a classifier that reference to other distinct classifiers, (iii) *Polymorphism*, i.e., number of abstract classifiers, (iv) *Complexity*, i.e., average number of features in a classifier and (v) *Design size*, i.e., number of classifiers. The calculated value of the understandability metric is negative which means the lower the value the more difficult is it to understand a metamodel.

We could improve the understandability of extracted compared to extensively reduced metamodel snippets in average by $\approx 26\%$. Considering the UML experiment, the understandability value of the UML metamodel is ≈ -61 whereas with the *Extract* operator we could improve this value to ≈ -12 when the focus is on UML's data modeling capabilities. With the restructuring of extracted metamodel elements, we could further improve the understandability by $\approx 48\%$ in the metamodel snippet.

5 Related Work

With respect to our goal of generating metamodel snippets, we identified three lines of related research work: (i) model slicing, (ii) model refactoring and (iii) aspect mining.

Model Slicing. Inspired from the notion of program slicing [Wei81], a static slicing mechanism is proposed by [KMS05] for UML class diagrams and by [BLC08] for modularizing the UML metamodel. Both approaches present how (meta)model elements are selected by relying on user-defined criteria (e.g., classifiers or relationships to be included) that express the initial set of elements from which a slice is computed. This computation is in our work supported by the *Extract* operator. Slicing mechanisms specific to UML class diagrams and state machine diagrams are introduced by [LKR10, LKR11]. In this research endeavor, class invariants and pre- and post-conditions of operations are exploited for computing class diagram slices while data and control flow analysis techniques are applied to reduce state machines to the elements relevant to reach a certain state. Since in our approach metamodels are solely considered from a structural viewpoint, techniques related to behavioral viewpoints (e.g., operational semantics) are beyond the scope of our approach. Slicing metamodels is addressed by approaches presented in [SMBJ09] and more recently in [KMG11]. They apply a projection-based approach to obtain a strictly structure-preserving subset as opposed to our approach that enables restructuring of metamodels. A declarative language as a means to implement slicing mechanisms for reducing (meta)models is introduced by [BCBB11] which could be an alternative technology to realize our *Reduce* operator.

Model Refactoring. Existing research work in the area of model refactoring is presented by [FGSK03] addressing pattern-based refactorings on UML-based models, [Wac07] proposing model refactorings for co-adapting models with evolved metamodels, or [MMBJ09] applying generic model refactorings on different kind of models. Our approach focuses on the metamodel level. We adopted commonly known refactorings originating from the area of object-orientation [Opd92, HVW11] to achieve type-safe metamodel reductions.

Aspect Mining. Since aspect-orientation has arrived at the modeling level, several research endeavors started addressing this topic as surveyed in [WSK$^+$11]. Identifying aspects in existing models is investigated by [ZGLT08], presenting approaches for mining crosscutting concerns in a given set of models and describing them with an aspect language. Metamodel snippets can be compared with the notion of symmetric concerns [HOT02, WSK$^+$11] since they subsume model elements related to certain modeling concerns.

6 Lessons Learned

We now summarize lessons learned from realizing and applying our proposed approach.
Type-safe restructuring as enabler for less complex metamodel snippets. Strictly structure-preserving approaches are not necessarily the first choice for generating metamodel snippets since restructuring can reduce the number of metamodel elements. Approaches that facilitate to compose (cf., [WS08]), combine (cf., [Val10]) or extend (cf., [LWWC12]) existing metamodels may benefit from reduced metamodel snippets if they intend to operate on a subset of a large metamodel.
Usage of existing metamodel implementation code for metamodel snippets. Decoupling metamodel snippets from their base metamodel requires care if existing metamodel implementation code is available. Clearly, this depends on the metamodeling workbench. We realized delegation mechanisms that loosely couple metamodel snippets with their base metamodel at the implementation level to cope with this challenge.
Metamodel snippets as reusable assets for new metamodels. Since reuse allows exploiting domain knowledge already expressed in existing metamodels [KKP+09], metamodel snippets can support reuse when only some parts of an existing metamodel are required. For instance, the language workbench challenge 2012[5] refers to such a reuse scenario. Clearly, metamodel snippets are a first step in this direction and may act as stimulator to enhance reuse in metamodeling.

References

[BCBB11] Arnaud Blouin, Benot Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling Model Slicers. In *MODELS'11*, pages 62–76. Springer, 2011.

[BD02] Jagdish Bansiya and Carl G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, 2002.

[BLC08] Jung Ho Bae, KwangMin Lee, and Heung Seok Chae. Modularization of the UML Metamodel Using Model Slicing. In *ITNG'08*, pages 1253–1254. IEEE, 2008.

[FGSK03] R. France, S. Ghosh, E. Song, and D.K. Kim. A metamodeling approach to pattern-based model refactoring. *IEEE Softw.*, 20(5):52–58, 2003.

[GCD+12] Clément Guy, Benoît Combemale, Steven Derrien, Jim R. H. Steel, and Jean-Marc Jézéquel. On model subtyping. In *ECMFA'12*, pages 400–415. Springer, 2012.

[HOT02] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Research Report RC22685, IBM, 2002.

[HVW11] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *SLE'11*, pages 163–182. Springer, 2011.

[5]http://www.languageworkbenches.net

[KKP$^+$09] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. In *OOPSLA'09 Workshop on Domain-Specific Modeling (DSM'09)*, 2009.

[KMG11] Pierre Kelsen, Qin Ma, and Christian Glodt. Models within Models: Taming Model Complexity Using the Sub-model Lattice. In *FASE'11*, pages 171–185. Springer, 2011.

[KMS05] Huzefa Kagdi, Jonathan I. Maletic, and Andrew Sutton. Context-Free Slicing of UML Class Models. In *ICSM'05*, pages 635–638. IEEE Computer Society, 2005.

[LKR10] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In *MODELS'10*, pages 228–242. Springer, 2010.

[LKR11] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Slicing Techniques for UML Models. *JOT*, 10:1–49, 2011.

[LWWC12] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *JOT*, 11(1):1–29, 2012.

[MMBJ09] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic Model Refactorings. In *MODELS'09*, pages 628–643. Springer, 2009.

[MSJ04] Haohai Ma, Weizhong Shao, Lu Zhang 0023, and Yanbing Jiang. Applying OO Metrics to Assess UML Meta-models. In *UML'04*, pages 12–26. Springer, 2004.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, 1992.

[SMBJ09] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model Pruning. In *MODELS'09*, pages 32–46. Springer, 2009.

[SMM$^+$12] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jézéquel. Reusable model transformations. *SoSym*, 11(1):111–125, 2012.

[Val10] Antonio Vallecillo. On the Combination of Domain Specific Modeling Languages. In *ECMFA'10*, pages 305–320. Springer, 2010.

[VG12] Antonio Vallecillo and Martin Gogolla. Typing Model Transformations Using Tracts. In *ICMT'12*, pages 56–71. Springer, 2012.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *ECOOP'07*, pages 600–624. Springer, 2007.

[Wei81] Mark Weiser. Program slicing. In *ICSE'81*, pages 439–449. IEEE Press, 1981.

[WS08] Ingo Weisemöller and Andy Schürr. Formal Definition of MOF 2.0 Metamodel Components and Composition. In *MODELS'08*, pages 386–400. Springer, 2008.

[WSK$^+$11] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elisabeth Kapsammer. A survey on UML-based aspect-oriented design modeling. *ACM Comput. Surv.*, 43(4):1–33, 2011.

[ZGLT08] Jing Zhang, Jeff Gray, Yuehua Lin, and Robert Tairas. Aspect mining from a modelling perspective. *Int. J. Comput. Appl. Technol.*, 31:74–82, 2008.

Automata-Based Refinement Checking for Real-Time Systems

Christian Brenner, Christian Heinzemann,
Wilhelm Schäfer
Software Engineering Group
Heinz Nixdorf Institute, University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
[cbr|c.heinzemann|wilhelm]@uni-paderborn.de

Stefan Henkler
OFFIS
Escherweg 2
26121 Oldenburg, Germany
stefan.henkler@offis.de

Abstract: Model-driven development of real-time safety-critical systems requires to support refinement of behavioral model specifications using, for example, timed simulation or timed bisimulation. Such refinements, if defined properly, guarantee that (safety and liveness) properties, which have been verified for an abstract model, still hold for the refined model. In this paper, we propose an automatic selection algorithm selecting the most suitable refinement definition concerning the type of model specification applied and the properties to be verified. By extending the idea of test automata construction for refinement checking, our approach also guarantees that a refined model is constructed correctly concerning the selected and applied refinement definition. We illustrate the application of our approach by an example of an advanced railway transportation system.

1 Introduction

Innovation in embedded real-time systems is increasingly driven by software [SW07]. Since embedded real-time systems often operate in safety-critical environments, errors in the software may cause severe damages. Thus, ensuring correct operation and safety of the software is mandatory, but challenging due to its high complexity. High complexity is not only a result of the complexity of the system in terms of its size but in addition, due to its strict real-time requirements. Embedded real-time systems require the system (and all its components) to produce the expected (correct) output no later than at a given point in time.

Model-driven software development addresses these challenges by building formal models of the software instead of implementing it directly. These models can be used to verify safety and liveness properties of the system under development using model checking [BK08]. For embedded real-time systems, timed automata [AD94, BY03] have proven to be a suitable model to support model checking [ACD93, BY03]. Model checking, however, does not scale for large systems. Therefore, an underlying component model is defined in such a way that it supports compositional verification, i.e. parts of the model are verifiable independently.

In a little more detail, components, e.g. RailCab and TrackSection in Figure 1, communicate via protocols (specified by timed automata) which define a sequence of timed message exchanges. Message exchange, in turn, is using connectors and, in case of asynchronous communication, buffers for storing incoming messages. Connector and buffer behavior is also specified using timed automata. A so-called abstract model of such a protocol that includes a model of the connector and possibly message buffers is verified to prove that it fulfills a given (safety) property φ. Then, the abstract protocol behavior is assigned to a component and usually refined according to the needs and context of the individual component. Note, that abstract protocols are defined in such a way that they become reusable in various contexts and for various components, possibly even in different systems.

A common approach is to check such a refinement for correctness rather than verifying φ for the refined protocol behavior again. Checking refinement for correctness is guaranteeing a correct refinement according to the definitions as given in Section 3. Such an approach makes formal verification of distributed systems, whose subcomponents communicate via protocols, a lot more scalable. Buffer and connector behavior specifications (e.g. by timed automata) do not need to be taken into account anymore when the refinement of protocols is checked for correctness.

However, a number of different refinement definitions have been proposed in the literature. Depending on the particular type of protocol which is refined, they might all be useful when building a system. In general, a refinement definition needs to be as weak as possible for enabling reuse of an abstract protocol in as many different contexts as possible, but as strong as necessary for guaranteeing that φ holds for the refined protocol behavior. That is especially useful if the same abstract behavior is used multiple times in the same system.

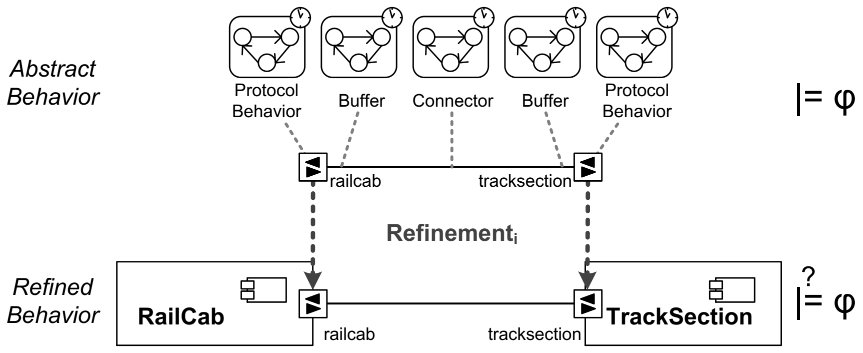


Figure 1: Overview of the Refinement Approach

Currently, the selection of a suitable refinement definition is left to the developer and his expertise without giving him any further tool support. If the developer selects a too weak refinement definition, it is not guaranteed that φ holds for the refined protocol behavior. If the selected refinement definition is too strong, the refinement check might reject the refined protocol behavior although it fulfills φ. This may happen, e.g., if the refined model removes behavior that is irrelevant for φ, but which is checked by the too strong refinement definition. In this paper, we provide an automatic selection algorithm selecting a suitable refinement definition based on the type of model, for example timed vs. untimed, as well as the specification of the property φ. As a basis, we identify the commonalities and differences of the six most relevant existing refinement definitions for distributed real-time systems.

The main contribution of this paper is an algorithm to support automatic checking of a correct protocol refinement based on an extension of the approach described in [JLS00]. In [JLS00], a so-called test automaton is automatically constructed to verify correct refinements. The test automaton encodes both, the abstract model and the constraints of the selected refinement. The constraints specify the allowed deviation of the refined system model from the abstract model. If (and only if) the refined system model violates one of the constraints, the test automaton enters a special error location indicating that the refinement is not correct. However, that approach is restricted to just one refinement definition (namely timed ready simulation which is explained below). Our extensions of [JLS00] provide for the construction of test automata for all the mentioned different refinement definitions. They include, in particular, the notion of asynchronous communication via buffers and thus a very important type of communication and corresponding refinement definition for distributed real-time systems. Checking refinement definitions for that case has not been considered before.

In this paper, we will use the RailCab system[1] as a case study for an embedded real-time system. The vision of the RailCab project is a railway transportation system where autonomous vehicles, called RailCabs, travel on existing track systems. Since RailCabs operate autonomously, collision avoidance on track has to be realized by software, only. For avoiding collisions, each RailCab must register at track sections for gaining admission before entering. This communication is safety-critical and must obey real-time requirements to ensure that a RailCab comes to a stop before entering a track if it has no admission. In our case study, we show how the same abstract behavior can be refined for four different types of track sections. Each type of track section requires the abstract behavior to be refined differently. Using our approach, we succeeded in showing the correctness of the refinements by using different refinement definitions for the different types of track sections.

The paper is structured as follows: In Section 2, we introduce timed automata. Section 3 presents the most relevant refinement definitions and the corresponding selection criteria using the RailCab example. The construction of the test automaton is given in Section 4. We discuss related work in Section 5 before concluding the paper in Section 6.

2 Timed Automata

In our approach, we use timed automata as a behavior model for the components within a system. They extend finite automata by a set of real-valued clocks [AD94, BY03]. Clocks measure the progress of time in a system and allow for the specification of time-dependent

[1]http://www.railcab.de

behavior. In essence, that means that the output of the automaton does not only depend on its inputs, but also on the points in time at which the inputs are received.

Based on its clocks, a timed automaton specifies time guards, clock resets, and invariants. A time guard is a clock constraint that restricts the execution of a transition to a specific time interval. A clock reset sets the value of a clock back to zero while a transition is fired. Invariants are clock constraints associated with locations that forbid that a timed automaton stays in a location when the clock values exceed the value of the invariant. In combination, time guards and invariants define the time intervals where transitions may fire at run-time.

In addition to guards and resets, transitions may carry messages that specify inputs and outputs of the timed automaton. Input messages are denoted by ?, output messages by !. We assume an asynchronous communication of the timed automata. Messages are sent over a connector and put into a buffer on the receiver side as shown in Figure 1. By providing explicit timed automata for the buffers and the connector, we map the asynchronous communication of our timed automata to the synchronous communication of timed automata as used in [BY03].

In timed automata, transitions are not forced to fire instantaneously if they are enabled. Instead, the automaton may rest in a location and delay. For many applications, this is not sufficient because they require that transitions fire immediately if a certain message has been received. As a result, transitions can be marked as *urgent*. If an urgent transition is enabled, it fires immediately without any delay [BGK^{+}96].

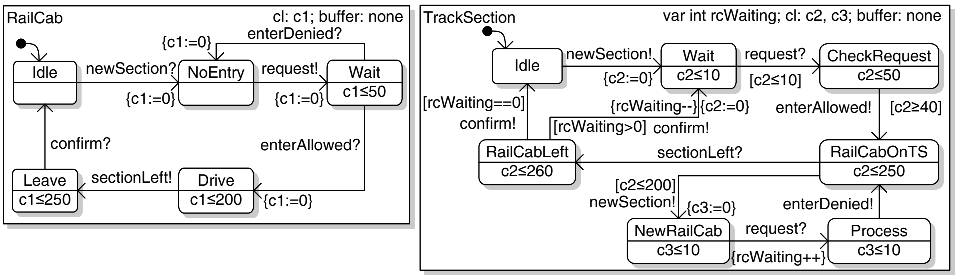


Figure 2: Abstract Behavior for Entering a Track Section

Figure 2 shows an example of two timed automata. They specify the protocol behavior of the abstract model of Figure 1. The automata specify a simplified registration protocol where RailCabs register at a track section to be allowed to enter it. Initially, both automata are in the Idle locations. Then, the track section sends newSection to an approaching RailCab. The RailCab requests to enter the track section by sending a request. The request is received by the track section which sends enterAllowed. Then, the RailCab switches to *Drive* and enters the track section. If another RailCab approaches, the track section sends newSection and switches to NewRailCab. In this case, it denies the request by sending enterDenied. The track section uses the variable rcWaiting to store the number of RailCabs waiting for entry. A RailCab finally sends sectionLeft after it left the track section which is confirmed by the track section. If there are RailCabs waiting, the track section switches to Wait for processing the next request. Otherwise, it switches to Idle.

The interaction of RailCabs and a track section is safety-critical, because RailCabs may come into collision if a RailCab enters a track section after the track section denied the entry. However, we also want to ensure that RailCabs are actually allowed to enter the track section. We use model checking to prove that the model fulfills such safety and liveness properties which ensure correct behavior. In the example, we need to verify two properties. First, we verify "If a track section sends enterDenied, then the RailCab will not send sectionLeft until the track section sends enterAllowed". If sectionLeft occurred before, this would imply that the RailCab entered the track section without being allowed to do so. Second, we verify "In all system states, there exists a path where the track section eventually sends enterAllowed." for checking that progress is possible.

We specify such properties formally by using the timed computation tree logic (TCTL, [ACD93]) and verify them using model checking. In TCTL, the first property is formalized as follows:

$$AG(enterDenied \Rightarrow A(\neg sectionLeft\ W\ enterAllowed)) \tag{1}$$

AG denotes that the formula in parentheses holds globally in all states of all execution paths. An occurrence of enterDenied implies that on all execution paths sectionLeft does not occur ($\neg sectionLeft$) until enterAllowed occurs which is modeled by AW. The property uses a so-called weak until (W) [BK08, pg. 327]. In contrast to the normal until (U) it does not require enterAllowed to occur eventually. A weak until, however, can be mapped to the standard TCTL operators [BK08, pg. 327].

The second property is formalized by:

$$AG(EF\ enterAllowed) \tag{2}$$

The operator EF denotes that enterAllowed is eventually sent.

3 Refinement Definition and Selection

A refinement definition relates an abstract model and a refined model of the same system as shown in Figure 1. It defines how the behavior defined by the refined model may deviate from the behavior defined by the abstract model. A restrictive refinement definition guarantees that verified safety and liveness properties still hold for the refined model. A less restrictive refinement definition leaves developers more flexibility to adapt the abstract model to a component and, thus, allows for more possible refined models. Finding a suitable refinement definition is, thus, a trade-off between flexibility upon building the refined model and properties that are preserved by the refined model.

In this section, we explain the six most relevant refinement definitions for embedded real-time systems informally due to space restrictions. Those are simulation [BK08], bisimulation [BK08], timed simulation [WL97], timed bisimulation [WL97], timed ready simulation [JLS00], and relaxed timed bisimulation [HH11]. For the informed reader, please note that we only consider so-called *weak* variants of the refinements [WL97]. These weak refinements abstract from any internal behavior which is defined by transitions not carrying a message, but performing an internal computation. It is sufficient to consider only weak variants because the protocol specifications, which are the subject of this paper, only specify message exchange between components.

Simulation requires that the refined model only includes sequences of messages that are specified already by the abstract model. The refined model, however, may remove sequences of messages. Thus, simulation preserves any CTL*-formulas [BK08] only containing $\forall$-path quantifiers. Formulas with an $\exists$-path quantifier are not preserved because the path fulfilling the property might be removed. For preserving CTL*-formulas with $\exists$-path quantifiers, we use bisimulation. It requires that the refined model includes exactly the same sequences of messages as the abstract model.

For timed automata, variants of simulation and bisimulation have been developed that impose conditions on the timing of messages. These conditions are absolutely necessary to refine protocols of real-time systems. Like the (untimed) simulation, variants of a timed simulation only preserve properties containing $\forall$-path quantifiers while variants of a timed bisimulation also preserve properties containing $\exists$-path quantifiers.

Timed simulation [WL97] requires that the refined model only includes sequences of messages that are specified already by the abstract model. In addition, the refined model only specifies sending or receiving a message in the same or a restricted time interval. If the abstract model uses urgent transitions, timed simulation is not sufficient. As shown in [JLS00], timed simulation does not guarantee that if the refined model R simulates an abstract model A, the composition with any other model B, $R \parallel B$, simulates $A \parallel B$. As a solution, [JLS00] presents a new refinement definition: the *timed ready simulation*. In addition to the conditions of a timed simulation, it requires the refined model to preserve all urgent transitions including their timing.

A timed bisimulation requires that the refined model includes exactly the same sequences of messages and specifies exactly the same time intervals as the abstract model. Therefore, timed bisimulation is a very strong refinement definition and preserves all TCTL properties. Using an input buffer for messages allows to relax the conditions of timed bisimulation. We call this relaxation *relaxed timed bisimulation* [HH11]. The relaxed timed bisimulation enables to extend the time intervals for received messages, but requires that the upper bounds of time intervals for sending messages are exactly the same as in the abstract model. It preserves all CTL*-formulas and all TCTL formulas only referring to the latest sending of messages.

For illustrating the selection of a refinement definition, we provide examples of two refinements of the abstract track section behavior of Figure 2. Figure 3 shows the behavior of a railroad crossing on the left and the behavior of a normal track section on the right. In addition, the RailCab system contains switches and stations which also execute refined versions of the abstract track section behavior of Figure 2. We omit the behavior descriptions for switches and stations due to space restrictions.

Informally speaking, the two automata specify the following behavior: If a RailCab wants to enter a railroad crossing, the railroad crossing must close the gates. The transition from

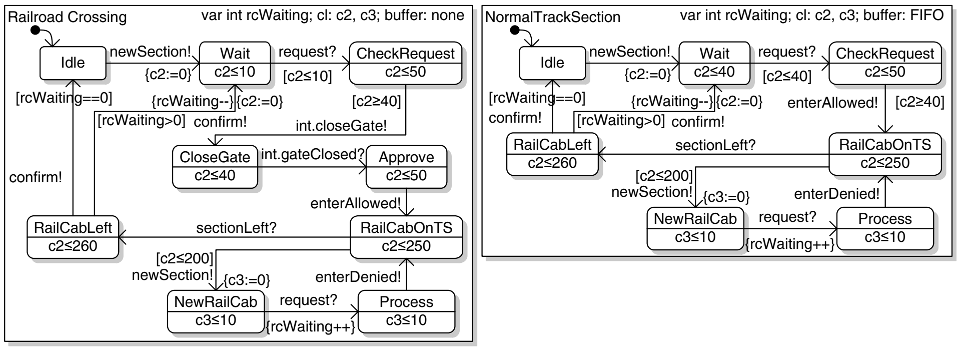


Figure 3: Refined Behavior for Railroad Crossings and Normal Track Sections

CheckRequest to RailCabOnTS is split into several transitions and intermediate locations that close the gates by using an internal message closeGate prefixed by int. After the gate responds that it is closed (gateClosed), the railroad crossing switches to Approve. Then, it sends enterAllowed and enters the RailCabOnTS location. In case of a normal track section, we only need to check whether the track is free. That, however, does not take as long as closing the gates at a railway crossing. Therefore, we may receive the input message later in a refined behavior utilizing the input buffer. Figure 3 shows the refined behavior for a normal track section. We relax the time guard of transition Wait to CheckRequest to $c2 \leq 40$. The remaining behavior remains unchanged.

After specifying the refined behavior models of Figure 3, we need to choose a suitable refinement definition for checking for a correct refinement. The choice of a suitable refinement definition depends on the verified properties to preserve as well as the characteristics of the (timed) automata used to model the system. Figure 4 summarizes the selection algorithm in form of a decision tree.

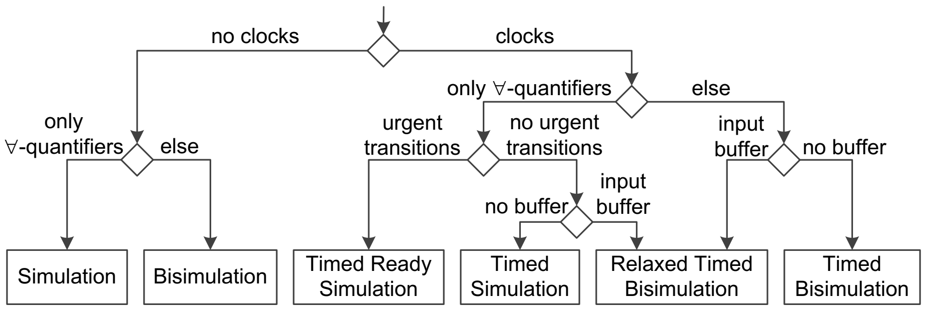


Figure 4: Decision Tree for Selecting a Refinement Definition

We can extract the necessary information for deriving a decision based on the decision tree by a syntactical analysis of both, the properties and the automata. For the first decision in the tree, we need to analyze whether the automata use clocks or not. Second, we check whether the properties only contain ∀-path quantifiers as, e.g., Property 1 in Section 2, or whether they also contain ∃-path quantifiers as, e.g., Property 2. Third, we analyze whether the abstract automaton uses urgent messages. Finally, we need the information whether the automata use a buffer for incoming messages provided by the developer.

In our example of Figure 3, the two refined automata need to preserve Properties 1 and 2 of Section 2. As a result, the decision algorithm selects the timed bisimulation for the refined model of the railroad crossing and it selects the relaxed timed bisimulation for the refined model of the normal track section. According to these definitions, the refinements given in Figure 3 are correct.

4 Test Automata Construction

Test automata have been introduced in [JLS00] as an approach for verifying refinements for timed automata. The basic idea of this approach is to encode the abstract model and the conditions for a correct refinement as a timed automaton T_A, called test automaton (Figure 5). These conditions define whether the developer is allowed to extend or restrict the time intervals for communication, or even to completely remove message sequences. Test constructs in T_A encode which changes are allowed and which are not, according to the conditions of the particular refinement definition (cf. Section 3).

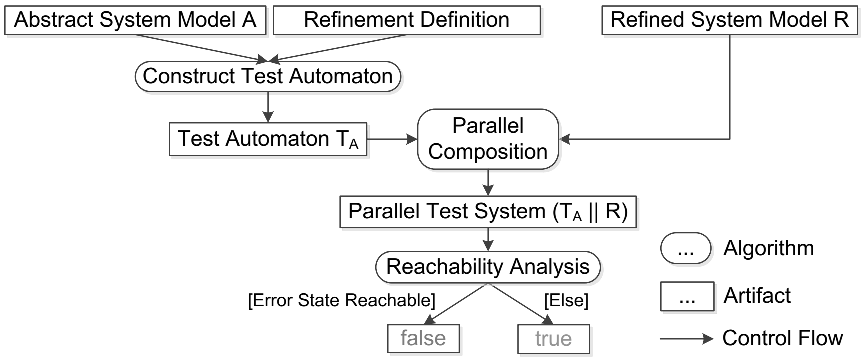


Figure 5: Verification using Test Automata

For the verification, we build the composed model $T_A \parallel R$, i.e. the parallel composition of T_A with the refined model R [BY03]. Then, we perform a reachability analysis on $T_A \parallel R$. During the reachability check, T_A communicates with the refined automaton R for detecting disallowed deviations from the message sequences of the abstract automaton A. If the conditions of the refinement definition are not fulfilled, the special error location Err in T_A becomes reachable. Otherwise, the refinement is correct.

Our test automaton construction generalizes and extends the construction as given in [JLS00]. The original construction only checks for a timed ready simulation. Our approach, on the contrary, supports checking all six refinement definitions given above (Figure 4). We extend the original approach of [JLS00] by introducing additional test constructs. This section explains how to construct T_A such that Err becomes reachable in $T_A \parallel R$ iff the selected refinement definition is *not* fulfilled for A and R.

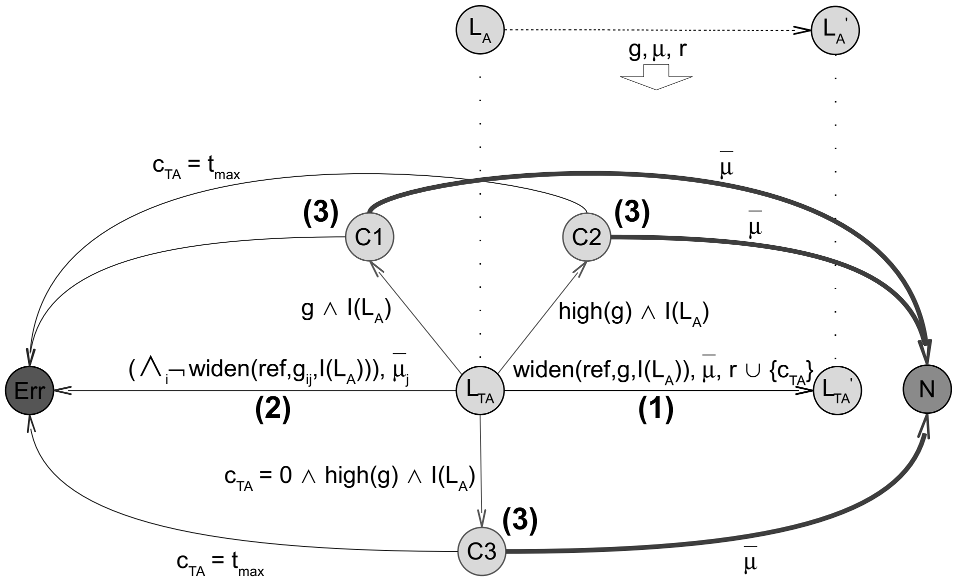


Figure 6: Construction Schema for our Test Automata

Figure 6 presents the schema for the construction of the part of T_A which is derived from a single transition $L_A \longrightarrow L_A{}'$ in the abstract model. T_A contains three kinds of test constructs, marked with (1)-(3) in Figure 6. These are explained in the following.

First, T_A must include all sequences of messages as defined by A, because all refinement definitions allow these sequences to be included in R. To model this in T_A, we define a corresponding transition $L_{TA} \longrightarrow L_{TA}{}'$ (1) for each transition $L_A \longrightarrow L_A{}'$ in A. The transition labels will be explained below.

Second, transitions $L_{TA} \longrightarrow Err$ (2) are defined for all sent and received messages which are not specified at outgoing transitions of L_A in A. One such transition is defined for each time interval in which a given message cannot be sent or received by A in L_A. If the developer refined A to R by adding communication not allowed by the refinement definition, these transitions make Err reachable. This so-called forbidden behavior must be checked for all refinement definitions, as none of them allows to add completely new message sequences in R (cf. Section 3).

Third, all variants of bisimulation (cf. Section 3) require that all sequences of messages specified in A are still included in R. Also, timed ready simulation requires all urgent communication in A to still be included in R. For checking this so-called required communication, T_A includes up to three locations $C1$, $C2$, and $C3$ (for each transition $L_A \longrightarrow L_A{}'$) (3). Transitions $L_{TA} \longrightarrow CX$, $CX \longrightarrow N$, and $CX \longrightarrow Err$ are established for each of these locations $CX(X \in [1, 2, 3])$. If the developer refined A to R by removing message sequences, T_A can reach Err via $C1$, $C2$, or $C3$. Successful tests for required communication lead to location N via $C1$, $C2$, or $C3$. Note that reaching N only indicates a successful test for one particular transition and does not allow any further conclusions about the correctness of the refinement.

The labels of the three test constructs for T_A depend on the specific refinement definition to check. We explain these labels in the following. We refer to [Bre10] for further technical details of the construction.

(1) Label definitions for allowed communication We create the labels for the transitions $L_{TA} \longrightarrow L_{TA}{}'$, modeling the message sequences allowed in R, as follows. Compared to the corresponding abstract transition $L_A \longrightarrow L_A{}'$, we invert the direction of all messages, i.e., input becomes output and vice versa. The symbol μ refers to the original message, $\overline{\mu}$ denotes the inverted one. This inversion ensures that T_A and R can synchronize in $T_A \parallel R$ whenever a sequence of messages is specified in both. Note that we use the $\parallel$-operator of UPPAAL[BY03].

Untimed simulation and bisimulation, as well as relaxed timed bisimulation each allow R to extend the time intervals defined in A. The time guard of each transition $L_{TA} \longrightarrow L_{TA}{}'$ in T_A is extended or removed accordingly by the function $widen$ (Figure 6). Depending on the refinement definition to check, given by ref in (Figure 6), $widen$ returns a modified time guard as follows. For untimed simulation and bisimulation, $widen$ returns a time guard that is always true, because time plays no role for these refinement definitions.

For relaxed timed bisimulation, the time guard returned by $widen$ depends on the transition $L_A \longrightarrow L_A{}'$. If $L_A \longrightarrow L_A{}'$ carries an input message, the returned time guard is always true. For these transitions, the relaxed timed bisimulation allows to extend time intervals arbitrarily. If $L_A \longrightarrow L_A{}'$ carries an output message, the returned time guard is the maximum of the upper bound of the time guard g of $L_A \longrightarrow L_A{}'$ and the invariants of L_A. This time guard models the condition of relaxed timed bisimulation that messages in R may not be sent later than in A. Earlier sending, however, is permitted by this time guard.

For the other three refinement definitions (timed simulation, timed bisimulation, and timed ready simulation), $widen$ returns the original time guard g, intersected with any invariants of L_A. This time guard defines the same time interval as specified for the abstract transition $L_A \longrightarrow L_A{}'$, because these refinements do not permit R to extend any time intervals of A.

(2) Label definitions for forbidden communication For each message μ_j in the alphabet of A which *is not sent or received* by any outgoing transition $L_A \longrightarrow L'_A$ in L_A, one transition $L_{TA} \longrightarrow Err$ is defined (cf. Figure 6). These transitions check for additional messages in R, which are not defined by A in the current location L_A. We need this check for all refinements, because none allows adding additional message sequences. The transition defined for a message μ_j carries the inverted message $\overline{\mu_j}$. This transition synchronizes with R, if R sends or receives the forbidden message μ_j. Then, Err becomes reachable in $T_A \parallel R$. We define the time guard of $L_{TA} \longrightarrow Err$ to be always true, because R may never offer μ_j, regardless of time.

For each message μ_j in the alphabet of A, which *is sent or received* by an outgoing transition $L_A \longrightarrow L'_A$ in L_A ($\mu = \mu_j$ in Figure 6), further transitions $L_{TA} \longrightarrow Err$ are defined. These transitions check whether the time intervals for sent or received messages in R are extended in comparison to the time intervals defined in A. We need this check for all timed refinements, because they do not allow R to extend the time intervals of A. Relaxed timed bisimulation allows extended time intervals in R, but forbids later sending of messages. To determine the time intervals where R may not define μ_j, we consider all transitions with μ_j in L_A. We write g_{ij} to refer to the time guard of the i-th transition with message μ_j in L_A. We create the conjunction of the negations of all these guards g_{ij}, each one modified by $widen$ (see above). The result are those time intervals in which μ_j is not defined in L_A. One transition $L_{TA} \longrightarrow Err$ is defined for each of these intervals. Each transition defined for μ_j carries the inverted message $\overline{\mu_j}$. If R sends or receives the forbidden message μ_j when no transition defining μ_j in L_A is enabled in A, R can synchronize with one of the transitions $L_{TA} \longrightarrow Err$ in T_A. Then, Err becomes reachable in $T_A \parallel R$.

(3) Label definitions for required communication $C1$ checks whether a message μ which is defined in A by a transition $L_A \longrightarrow {L_A}'$ is also defined by R during the time interval in which $L_A \longrightarrow {L_A}'$ is enabled. We need this check for timed bisimulation and timed ready simulation, because they do not allow R to restrict the time intervals for messages that were defined in A. For the transition $L_{TA} \longrightarrow C1$, we take over the time guard g of $L_A \longrightarrow {L_A}'$ and intersect it with the invariant $I(L_A)$ of the location L_A. This makes $C1$ reachable during the time interval in which $L_A \longrightarrow {L_A}'$ is enabled. The urgent transition $C1 \longrightarrow N$ carries the message $\overline{\mu}$. Whenever R can send or receive μ, it synchronizes with $C1 \longrightarrow N$, leading to N in $T_A \parallel R$. For $C1 \longrightarrow Err$ we define no time guard. Err becomes reachable via $C1 \longrightarrow Err$ in $T_A \parallel R$, whenever $C1 \longrightarrow N$ is not enabled. Because $C1 \longrightarrow N$ is urgent, it has precedence over $C1 \longrightarrow Err$ and prevents it from triggering while R sends or receives μ. If at any point in the time interval $g \wedge I(L_A)$, in which $L_A \longrightarrow {L_A}'$ is enabled, R *does not* send/receive μ, Err becomes reachable via $C1$ in $T_A \parallel R$.

$C2$ checks whether a message μ which is sent in A by a transition $L_A \longrightarrow {L_A}'$ is also sent by R at the end of the time interval in which $L_A \longrightarrow {L_A}'$ is enabled, or later. We need this check for the relaxed timed bisimulation. This refinement requires that the upper bounds of time intervals for sending messages in R are exactly the same as in A. The test construct (2) already checks that these time interval bounds are not raised in R. $C2$ checks that they are also not lowered, i.e. R must be able to send μ *at least* up to the upper bound in A. The latest time at which $L_A \longrightarrow {L_A}'$ is enabled is defined by the upper bound $high(g)$ of the time guard g and the invariant $I(L_A)$ of L_A, whichever is more restrictive. To ensure that $C2$ is entered only up to this time, we set the time guard of $L_{TA} \longrightarrow C2$ to $high(g) \wedge I(L_A)$. The urgent transition $C2 \longrightarrow N$ carries the message $\overline{\mu}$ to synchronize with R if it defines μ. We set the time guard of $C2 \longrightarrow Err$ to $c_{TA} = t_{max}$, to only allow reachability of Err after a maximum amount of time t_{max} has passed. The value

t_{max} is chosen high enough not to be reached in any actual execution of the system. If R, after reaching $C2$, still sends μ, the urgent transition $C2 \longrightarrow N$ forces T_A to enter N in $T_A \parallel R$. Reachability of Err is prevented in this case. If R never sends μ after this time, $c_{TA} = t_{max}$ eventually becomes true. Then, Err becomes reachable via $C2$ in $T_A \parallel R$.

$C3$ checks whether a message μ which is defined in A by a transition $L_A \longrightarrow L_A{}'$ is also defined by R at an arbitrary time. We need this check for untimed bisimulation and (the untimed condition of) relaxed timed bisimulation. These refinements require message sequences defined by A to also be defined by R but do not restrict timing. As above, the urgent transition $C3 \longrightarrow N$ carries the message $\overline{\mu}$ to synchronize with R when R defines μ. In the time guard of $L_{TA} \longrightarrow C3$, we check $c_{TA} = 0$ to make $C3$ reachable only right after T_A entered L_{TA}. The special clock c_{TA} is reset with every transition $L_{TA} \longrightarrow L_{TA}{}'$. Checking $c_{TA} = 0$ ensures that the time interval, in which R may fulfill the check by defining μ, starts directly after the last message exchange. It can not be reduced by T_A entering $C3$ at a later time. As for $C2$, we intersect the time guard of $L_{TA} \longrightarrow C3$ with $high(g) \wedge I(L_A)$. This intersection ensures that $C3$ is not reachable after the latest time the abstract transition $L_A \longrightarrow L_A{}'$, defining μ, is enabled. This is the case when L_A is entered with clock values already higher than the time guard of $L_A \longrightarrow L_A{}'$. Then, R is not required to define μ either and $C3$ is not reachable. We set the time guard of $C3 \longrightarrow Err$ to $c_{TA} = t_{max}$ (see above). Err only becomes reachable in $T_A \parallel R$ if R never defines the message μ after T_A entered location L_{TA} by a previous synchronization. Otherwise, the urgent transition $C3 \longrightarrow N$ synchronizes with R and forces T_A into N.

Implementation We implemented the automatic generation of the test automata based on a given refinement definition as described in [Bre10]. In addition, we support to verify that the refinement holds based on the parallel test system $T_A \parallel R$.

5 Related Work

We discuss related work from two areas. First, we review related work on approaches that support multiple refinement definitions. Second, we discuss related work on test automata.

Reeves and Streader [RS08a, RS08b] identify commonalities and differences of refinement definitions for process algebras and unify them in a generalized definition, but provide neither a selection nor a verification algorithm. Sylla et al. [SSdR05] present a refinement definition program including a refinement check where the refinement is parameterized by a particular LTL formula [BK08] such that only this particular formula is preserved. In contrast to our approach, both do not consider real-time properties. In [Bey01], Beyer introduces timed simulation for Cottbus Timed Automata which are a special kind of timed automata. We cover this refinement definition in our refinement check.

Test automata are used by [ABBL03] for model checking temporal properties specified in SBBL (Safety Model Property Language) on timed automata rather than verifying correct refinements. The test automata construction follows the same idea of encoding the conditions for correctness into the states of an automaton. The generated test constructs, however, are different. The approaches [GPVW96] and [Tri09] perform LTL model checking [BK08] on (timed) Büchi automata and encode the properties in automata as well. Again, the construction differs from our approach.

6 Conclusion and Future Work

In this paper, we present an automated automata-based refinement check for timed automata. Based on the (timed) automata used to specify the abstract and refined model of the system and the verified properties, we automatically select the most suitable refinement definition out of a set of six refinement definitions. The most suitable refinement definition is the least restrictive refinement definition that preserves all verified properties. Then, we automatically generate a so-called test automaton which encodes the abstract model and the conditions of the corresponding refinement. Our construction extends the construction of [JLS00] by additional test constructs. Using the test automaton, we verify whether all relevant properties still hold for the refined model.

Our approach enables developers of real-time systems to reuse (abstract) verified models of protocols that are specified in terms of (timed) automata. By verifying the correctness of refinements, we ensure that all verified properties are preserved. We relieve the developer from choosing a suitable refinement definition by automatically identifying the most suitable refinement based on the given model and the verified properties.

Future works will investigate whether we can relax the restrictions that currently apply to our test automaton construction. At present, the construction only allows to check for a correct refinement of a single timed automaton. Checking refinements for networks of timed automata requires to build a product automaton for the network [BY03]. We plan to investigate how the construction can be extended such that the explicit construction of the product automaton is not necessary. Furthermore, we want to extend the presented construction of refinements and test automata to dynamic communication structures. In a dynamic communication structure, the concrete communication topology may change during run-time which requires timed automata to be instantiated and deinstantiated dynamically.

Acknowledgments This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft. Christian Brenner is supported by the International Graduate School Dynamic Intelligent Systems. We thank Matthias Becker, Steffen Becker, and Uwe Pohlmann for useful comments.

References

[ABBL03] L. Aceto, P. Bouyer, A. Burgueño, and K. Larsen. The power of reachability testing for timed automata. *Theor. Comput. Sci.*, 300(1-3):411–475, 2003.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104:2–34, 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[Bey01] D. Beyer. Efficient Reachability Analysis and Refinement Checking of Timed Automata Using BDDs. In T. Margaria and T. Melham, editors, *Proc. of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume 2144 of *Lecture Notes in Computer Science*, pages 86–91, 2001.

[BGK$^+$96] J. Bengtsson, D. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In R. Alur and T. A. Henzinger, editors, *CAV 96*, number 1102 in LNCS, pages 244–256. Springer, July 1996.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Bre10] C. Brenner. Analyse von mechatronischen Systemen mittels Testautomaten. Masterarbeit, Universität Paderborn, August 2010.

[BY03] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[GPVW96] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. of the 15th IFIP WG6.1 Intern. Sym. on Protocol Specification, Testing and Verification XV*, pages 3–18. Chapman & Hall, Ltd., 1996.

[HH11] C. Heinzemann and S. Henkler. Reusing Dynamic Communication Protocols in Self-Adaptive Embedded Component Architectures. In *Proc. of the 14th Intern. Sym. on Component Based Software Engineering*, CBSE '11, pages 109–118. ACM, June 2011.

[JLS00] H. E. Jensen, K. Larsen, and A. Skou. Scaling up Uppaal Automatic Verification of Real-Time Systems Using Compositionality and Abstraction. In *Proc. of the 6th Intern. Sym. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '00)*, pages 19–30. Springer, 2000.

[RS08a] S. Reeves and D. Streader. General Refinement, Part One: Interfaces, Determinism and Special Refinement. *Electron. Notes Theor. Comput. Sci.*, 214:277–307, June 2008.

[RS08b] S. Reeves and D. Streader. General Refinement, Part Two: Flexible Refinement. *Electron. Notes Theor. Comput. Sci.*, 214:309–329, June 2008.

[SSdR05] M. Sylla, F. Stomp, and W.-P. de Roever. Verifying parameterized refinement. In *Proc. 10th Intern. Conf. on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 313 – 321, june 2005.

[SW07] W. Schäfer and H. Wehrheim. The Challenges of Building Advanced Mechatronic Systems. In *Future of Software Engineering (FOSE '07)*, pages 72–84. IEEE, 2007.

[Tri09] S. Tripakis. Checking timed Büchi automata emptiness on simulation graphs. *ACM Trans. Comput. Logic*, 10:15:1–15:19, April 2009.

[WL97] C. Weise and D. Lenzkes. Efficient Scaling-Invariant Checking of Timed Bisimulation. In *Proc. of STACS'97, LNCS 1200:pages 177–188*, pages 177–188. Springer, 1997.

Über die Auswirkungen von Refactoring auf Softwaremetriken

Stefan Burger[1] & Oliver Hummel[2]

[1]Software-Engineering-Gruppe
Universität Mannheim
68131 Mannheim
sburger@mail.uni-mannheim.de

[2]Institut für Programmstrukturen und Datenorganisation
Karlsruher Institut für Technologie (KIT)
76128 Karlsruhe
hummel@kit.edu

Abstract: Softwaremetriken wurden in der Vergangenheit bereits vielfach diskutiert und auch kritisiert, werden aber in der industriellen Praxis nach wie vor als einfach zu erhebende Indikatoren für Codequalität oder gar die Qualität einer Software selbst angewendet. Gemessene Werte und Grenzwerte sind dabei oft schwierig zu interpretieren bzw. willkürlich festgelegt, so dass ihr Nutzen nach wie vor in Frage gestellt werden muss. Dieser Beitrag untermauert die bestehenden Kritiken an Softwaremetriken, indem er zunächst beispielhaft zeigt, wie Messergebnisse gängiger Metriken mit schlechter lesbarem aber funktional identischem Code künstlich „verbessert" werden können. Darauf aufbauend präsentiert er erste Analyseergebnisse (von Fowlers „Video-Store" und dem Lucene-Open-Source-Projekt), die darlegen, dass bekannte Softwaremetriken Code-Verbesserungen durch Refactoring nicht erkennen können, bzw. umgekehrt betrachtet, Refactoring die Codequalität verschlechtert. Ferner zeigen sich gängige Qualitätsmodelle ebenfalls weitgehend unempfindlich für die durchgeführten Refactorings.

1. Einleitung

Eine weit verbreitete Methode zur Messung der Wartbarkeit von Software wird unter dem Begriff statische Code-Analyse zusammengefasst [Ho08] und untersucht Quellcode mit Hilfe von Softwaremetriken, um mögliche Schwachstellen in dessen Qualität, wie beispielsweise eine hohe Komplexität, zu identifizieren. Bekannte, vor allem zur Analyse der Wartbarkeit und ihrer Untereigenschaften wie Verständlichkeit und Änderbarkeit [IO11] bereits seit Jahrzehnten genutzte Softwaremetriken sind beispielsweise McCabe's zyklomatische Komplexität [MC76], Halsteads Software Science Metrics [Ha77], oder auch jüngere objektorientierte Metriken, wie etwa die Kopplung zwischen Klassen [CK94]. Dabei gilt bisher vereinfacht gesagt meist die Grundannahme, dass „einfacherer" Code niedrigere Metrikmesswerte haben sollte (vgl. [Ha77, Mc76]), da kürzerer und somit weniger komplexer Code offensichtlich leichter zu verstehen ist [SA05, MR07]. Obwohl in der Literatur verschiedene, auf Metriken basierende

Qualitätsmodelle existieren (z.B. [PR10, AYV10]), sind direkte Nachweise einer Korrelation zwischen Metriken und Code- oder gar Softwarequalität bis dato nicht sehr zahlreich und konnten hauptsächlich für die Fehlerhäufigkeit gezeigt werden [NBZ06, SZT06]. Trotz dieser weitgehend fehlenden theoretischen Untermauerung werden in der Industrie oftmals Metrikwerte für Systeme definiert, die Auftragnehmer zu erreichen haben: zu liefernde Gewerke werden entsprechend auf die Einhaltung dieser Grenzwerte hin optimiert, in der Hoffnung damit auch die Systemqualität insgesamt zu verbessern. Es bleibt aber nach wie vor unklar, ob klassische Softwaremetriken tatsächlich in der Lage sind, gutes, d.h. verständliches und änderbares Softwaredesign zu erfassen. So fand Seng [Se07] bereits vor einigen Jahren heraus, dass eine automatisierte Code-Optimierung auf Basis von Metriken höherwertige Strukturen wie Design Patterns [GHJ10] zerstören kann. Auch vorangegangene Arbeiten der Autoren zur Analyse der Kabinensoftware eines Airbus-Flugzeugs mit Hilfe von Softwaremetriken [BH12] blieben im Hinblick auf Codequalität weitgehend aussagelos, da die verwendeten Metriken nicht in der Lage waren, hohe Domänenkomplexität von schlechter Programmierung zu unterscheiden.

Refactoring [Fo99] ist im Gegensatz zu Metriken ein recht junger Ansatz zur Verbesserung der Codequalität: sein Ziel ist es, auf Basis von funktionalitätserhaltenden Transformationen des Quellcodes, dessen Komplexität zu reduzieren und damit seine Lesbarkeit und Änderbarkeit zu verbessern. Dazu werden schlechte Programmierpraktiken, sog. „Code Smells", durch bessere bzw. lesbarere Konstrukte ersetzt, während gleichzeitig durch Regressionstesten sichergestellt wird, dass keine neuen Fehler in die Software eingeführt wurden. Code Smells und Softwaremetriken stellen somit beide eine wichtige Indikatorfunktion für mangelhaften Code bereit, allerdings wurden die Auswirkungen von Refactoring auf Softwaremetriken und eventuelle Zusammenhänge zwischen beiden bis heute nur wenig betrachtet. Erste Arbeiten von Du Bois et al. [DDV04] und Stroggylos und Spinellis [SS07] deuten aber bereits darauf hin, dass Refactoring einen eher negativen Einfluss, auf die Messwerte gängiger objektorientierter Softwaremetriken [CK94] haben kann.

Dieser Beitrag geht daher der Frage nach, wie die Messwerte der gängigsten Softwaremetriken durch eine gezielte Veränderung der Quelltextstruktur durch Refactoring beeinflusst werden. Unter der Annahme, dass planvoll durchgeführte Refactorings tatsächlich die Codequalität erhöhen, eignen sie sich offensichtlich ideal für einen Vorher-Nachher-Vergleich von Metriken, der neue Aufschlüsse über deren Aussagekraft bzgl. Codequalität geben könnte. In den folgenden Kapiteln wird sich auf Basis der dort vorgestellten Untersuchungen, die weit über bisherige Arbeiten hinausgehen, herausstellen, dass die von Fowler propagierten Refactorings Messwerte bekannter Softwaremetriken oftmals negativ beeinflussen bzw. einen überwiegend vernachlässigbaren Einfluss auf darauf aufbauende Qualitätsmodelle ausüben. Das folgende Kapitel liefert zunächst einen vertieften Einblick zu Softwaremetriken und Refactorings, bevor Kapitel 3 die von Fowler vorgeschlagenen Refactorings mit Hilfe von über 20 bekannten Metriken vermisst. Kapitel 4 erläutert die bei der Vermessung zweier Fallstudien (Fowlers „Video Store"-Beispiel und das quelloffene Lucene-Framework) gewonnenen Erkenntnisse, die in Kapitel 5 diskutiert und in Kapitel 6 zusammengefasst werden.

2. Forschungsstand zu Metriken und Refactorings

Um die vorwiegend kritische Haltung vieler Autoren gegenüber Softwaremetriken noch einmal zu unterstreichen, gibt dieses Kapitel zunächst einen kurzen Abriss vor allem von Arbeiten, die die Aussagekraft bekannter Softwaremetriken in Frage stellen. Danach werden wichtige Grundlagen des Refactoring vorgestellt und diskutiert, bevor eine Gegenüberstellung beider Ansätze die Kernidee dieses Beitrags, nämlich ein besseres Verständnis für die Beeinflussung von Metriken durch Refactorings, illustriert.

2.1 Softwaremetriken

Kritische Berichte zu Softwaremetriken sind in der Literatur mittlerweile recht zahlreich. Vor einigen Jahren entdeckten z.B. van der Meulen und Revilla [MR07] eine Abhängigkeit von McCabes zyklomatischer Komplexität (CC): sie konnten in ihren Untersuchungen nachweisen, dass diese stark von der Länge des betrachteten Quelltexts, also den Lines of Code (LOC) beeinflusst wird (mit einer Korrelation von 0,95), die Codekomplexität also offenbar mit der Länge des Programmcodes wächst. Kaur et al. [Ka09] untersuchten in ihrer Analyse sowohl McCabes CC als auch diverse Halstead-Metriken und kamen zu dem Schluss, dass die jeweiligen Definitionen einige schwerwiegende Probleme beinhalten. Beispielhaft genannt seien hier die nur vage Definition von Operanden und Operatoren bei Halstead oder abermals erkannte Abhängigkeiten der zyklomatischen Komplexität von der Programmlänge. Des Weiteren sollen an dieser Stelle Jones et al. [JC94] genannt werden, die in ihrer Arbeit versuchten, die Stärken und Schwächen verschiedener Metriken zu identifizieren. Auch in dieser Veröffentlichung wird auf Lücken in der Definition mehrerer Metriken hingewiesen, beispielhaft genannt seien sowohl die oft ungenaue Definition der LOC als auch die Einflüsse verschiedener Programmiersprachen – bzw. Programmierstile auf gängige Metriken mit LOC-Bezug.

Stamelos et al. [ST02] liefern eine ausführliche Fallstudie zur Anwendung von Softwaremetriken auf über 100 in C geschriebene Applikationen (z.B. mail) aus der SUSE 6.0 Linux-Distribution. Doch auch der Erkenntnisgewinn dieses Beitrags bleibt ernüchternd: zwar konnten die Metriken mit bekannten Grenzwerten (z.B. aus [Mc76, Ha77]) verglichen werden, doch eindeutige Qualitätsaussagen waren auf Basis dieses einfachen Modells nicht möglich. Ferner fanden sich bei Stamelos et al. verschiedene Ausreißer, die weit über den Grenzwerten lagen, ohne dass dafür konkrete Ursachen in Form von „schlechtem“ Code festzustellen waren, ähnlich wie das bei der zuvor bereits angesprochenen Untersuchung der Autoren der Fall war [BH12]. Wie bereits in der Einleitung angedeutet, konnten Nagappan et al. [NBZ06] und Schröter et al. [Sc06], jüngst immerhin einen Zusammenhang zwischen Komplexitätsmetriken und der Anzahl an Fehlern in Softwaresystemen empirisch nachweisen. Ein weiterer Nachweis der Anhängigkeit zwischen Coupling-Metriken und Softwarefehlern wurde von Binkley et al. [BS09] geführt. Es bleibt allerdings nach wie vor unklar, wie sich hohe Werte bei Komplexitätsmetriken auf andere Qualitätsfaktoren wie beispielsweise die Verständlichkeit des Quelltextes im Detail auswirken, auch wenn als erste Vermutung natürlich ein Zusammenhang zwischen hohen Komplexitätsmesswerten und schlechter Verständlichkeit naheliegend scheint.

2.2 Refactoring

Eine der ersten Arbeiten im Bereich Refactoring wurde 1992 von Opdyke [Op92] veröffentlicht. In seiner Dissertation werden u.a. erste Refactoring-Patterns zum Erzeugen von Super- und Unterklassen beschrieben. Die Grundidee von Refactoring ist es, bestehenden Code ohne Veränderung der Funktionalität neu zu strukturieren, um dadurch die Codekomplexität zu reduzieren und spätere Anpassungen und Erweiterungen einfacher durchführen zu können sowie dadurch Zeit und Kosten für Weiterentwicklungen zu reduzieren [Al01]. In seinem bekannten Buch aus dem Jahre 1999 [Fo99] fasst Fowler eine Reihe häufig in der Praxis genutzter Refactoring-Patterns und Anwendungsfälle zusammen. Eine Übersicht über jüngere Forschung im Bereich Refactoring liefern beispielsweise Mens et al. [MT04], die auch mögliche positive Auswirkungen von Refactoring auf einzelne Qualitätsindikatoren, wie Wiederverwendbarkeit oder Komplexität, beschreiben. Eine erste Analyse der Auswirkungen von Refactorings auf Softwarewartungen durch Wilking et al. [WKK7], zeigt immerhin eine leichte Verbesserung der Wartbarkeit restrukturierter Software, die allerdings mit einem zusätzlichen Aufwand für das Refactoring erkauft werden muss. Weitere empirisch gesicherte Erkenntnisse über die Auswirkungen von Refactoring auf die Code- oder gar Softwarequalität sind den Autoren bis dato nicht bekannt. Nichtsdestotrotz ist natürlich eine Hoffnung der Refactoring-Befürworter, dass eine regelmäßige Überarbeitung der Codestruktur zumindest einer Komplexitätserhöhung des Quelltexts entgegen wirkt.

2.3 Metriken und Refactoring

Wie bereits in der Einleitung angesprochen, sind den Autoren bisher nur drei Arbeiten geläufig, die objektorientierte Softwaremetriken vor und nach einem Refactoring vergleichen. Alshayeb [Al09] untersuchte die Auswirkungen von Refactoring auf fünf Software-Qualitätsindikatoren (adaptability, maintainability, understandability, reusability und testability). Dazu wurden die Auswirkungen von Refactorings auf einzelne Methoden dreier ausgewählter Programme mit Hilfe von neun Metriken (DIT, NOC, CBO, RFC, FOUT, WMC, NOM, LOC und LCOM [CK94]), vgl. Tabelle 3) analysiert und auf die Indikatoren übertragen. Die Untersuchung kam zu dem wenig greifbaren Ergebnis, dass das Zusammenspiel zwischen Refactorings und Qualitätsmerkmalen sehr komplex zu sein scheint und künftig genauer untersucht werden sollte.

In Du Bois [DM03] Untersuchungen eines einfach LAN-Simulator-Systems erhöhten (+) sich Messwerte der fünf genutzten objektorientierten Metriken nach einem Refactoring bei 50% der Ergebnisse (s. Tabelle 1). Bei 30% blieb das Ergebnis unverändert (o) und bei nur rund 20% sanken (-) die Messwerte.

Tabelle 1. Überblick über die Ergebnisse von Du Bois et al. [DM03]

Refactoring	Number of Methods	Number of Children	Coupling betw. Obj.	Response for a Class	Lack of Cohesion
EncapsulateField	+	o	o	+	+
PullUpMethod subclass	-	o	-	-	-
PullUpMethod superclass	+	o	+	+	+
ExtractMethod	+	o	o	+	+

Auch Stroggylos und Spinellis [SS07] kommen in ihren Stichproben auf teilweise deutliche Erhöhungen der Metrikergebnisse von bis zu 50 Prozent. Ihre Arbeit untersuchte ebenfalls nur objektorientierte Metriken (wie z.B. die in Tabelle 1 ebenfalls gezeigten) und kleinere, unabhängige Stichproben von Programmen wie Apache Log4j, MySQL Connector/J und JBoss und bleibt somit abermals von begrenzter Aussagekraft.

Zusammenfassend bleibt festzuhalten: bisherige Untersuchungen basierten zumeist auf wenigen Refactoring-Patterns und analysierten nur eine geringe Anzahl von Metriken (die objektorientierten nach Chidamber und Kemerer [CK94]). Ferner ist den Autoren bis dato keine Untersuchung bekannt, die die Auswirkungen von Refactorings beispielsweise auf die Halstead-Metriken [Ha77] oder über mehrere konsekutive Refactoring-Schritte hinweg analysiert hätte. Des Weiteren existiert noch keine detaillierte Analyse über den Einfluss aller von Fowler gelisteten Refactorings auf gängige Softwaremetriken, so dass nach wie vor unklar bleibt, in wie weit Beiden ein ähnliches Verständnis der Codekomplexität zugrunde liegt. Diese Arbeit leistet einen ersten Beitrag zum Schließen dieser Lücke, indem sie alle Fowler-Refactorings mit den in Literatur und Praxis gängigsten Softwaremetriken analysiert und zudem zwei Case Studies entsprechend untersucht.

2.4 Zur „Optimierung“ von Metriken

In der Industrie ist es („unter der Hand“) eine durchaus gängige Praxis, Softwaresysteme so zu „optimieren“, dass sie zuvor vereinbarte Grenzwerte bei Metriken einhalten können. Implizit einher geht damit der Wunsch, dass dadurch die Codekomplexität und auch die Softwarequalität verbessert werden mögen. Wie im Folgenden beispielhaft gezeigt, ist eine solche Verbesserung der Messwerte mit etwas Hintergrundwissen über den Aufbau der Metriken sehr leicht machbar, führt aber im folgenden Beispiel zu offensichtlichen Verschlechterungen in der Lesbarkeit des Quelltexts bzw. umgekehrt würde ein entsprechendes funktionserhaltendes und sinnvolles Refactoring des Codes zu schlechteren Metrikwerten führen. Alle im weiteren Verlauf vorgestellten Messergebnisse wurden mit dem kommerziellen Analysewerkzeug Understand[1] (Version 3.0) von Scientific Tools Inc. erhoben, so dass Beeinflussungen der Messungen durch eine unterschiedliche Interpretation der Metriken ausgeschlossen werden können.

Vorrangiges Ziel des folgenden einfachen Versuchs ist es, die Auswirkungen eines Refactorings auf Halsteads Softwaremetriken [Ha77] zu demonstrieren. Dazu wurde eine Berechnung der „Mitternachts-Formel“ zur Lösung quadratischer Gleichungen vereinfacht, indem die eigentliche Gleichung auseinander gezogen und mit Zwische-

[1] http://www.scitools.com (Letzter Zugriff 21.07.2012)

nergebnissen versehen wurde. Ein solches Vorgehen ermöglicht es Entwicklern üblicherweise die einzelnen Schritte leichter zu erfassen und zu verstehen und entspricht dem Refactoring-Pattern „Introduce Explaining Variable“ [Fo99]). Hier wurde die Formel in drei Teile aufgespalten und jedes Teil erhielt einen beschreibenden Namen (endResult für das Endergebnis oder qudEquPos für die positive Lösung der quadratischen Gleichung; die Abkürzungen wurden rein aus Platzgründen verwendet, in der Praxis sollten die Begriffe natürlich vollständig ausgeschrieben werden).

Tabelle 2. Funktional identische Code-Snippets mit unterschiedlichen Metrikwerten.

Snippet 1a	**Snippet 1b**

int result= -(b+sqrt(pow(b,2)-4*a*c)/2*a) -(b-sqrt(pow(b,2)-4*a*c)/2*a);	int qudEquPos = -(b+sqrt(pow(b,2)-4*a*c)/2*a);
	int qudEquNeg = -(b-sqrt(pow(b,2)-4*a*c)/2*a);
	int endResult = qudEquPos - qudEquNeg;

if(getCondition()<5)
if(getBill() != getCondition())
if(!(getEnable() != OCCUPIED))
condition = true;
	if(getCondition()<5) {
	if(getBill() != getCondition()) {
	if(!(getEnable() != OCCUPIED)) {
	condition = true;

Die verbesserte Lesbarkeit des Quellcodes wird also insbesondere durch das Einfügen zusätzlicher Variablen erreicht. Aus der Sicht der Halstead-Metriken benötigt der Code nun deutlich mehr Operatoren (z.B. +, =) und Operanden (z.B. qudEquPos), was automatisch zu schlechteren Metrikwerten führt (bspw. verschlechtert sich das Halstead Volume von 413 auf 493). Die zusätzlich eingefügten Variablen haben aber nicht nur einen negativen Effekt auf die Halstead-Metriken, durch das Refactoring evtl. überflüssig gewordene und entfernte Kommentare würden ferner den Kommentaranteil des Codes reduzieren, der oftmals ebenfalls als Qualitätsmerkmal [KWR10] angesehen wird. Bereits in diesem einfachen Beispiel ist die Robustheit der Metriken gegenüber einem einfachen Refactoring also offenbar nicht gegeben.

3. Analyse der Fowler-Patterns

Die eben beispielhaft gezeigten Einflüsse des Patterns „Introduce Explaining Variable“ auf bekannte Softwaremetriken werfen die Frage auf, ob auch weitere Refactorings, die die Lesbarkeit und Verständlichkeit von Sourcecode erhöhen sollen, von gängigen Softwaremetriken ebenfalls negativ bewertet werden würden? Fowler [Fo99] beschreibt in seinem bekannten Buch und auf seiner Webseite [Fo12] immerhin mehr als 50 solcher Patterns. Um die grundlegenden Wechselwirkungen zwischen Refactoring und Softwaremetriken zu analysieren, wurden für alle Refactorings jeweils die von Fowler gegebenen Beispiele mit den heute gängigsten Softwaremetriken (s. Tabelle 3) vermessen: einmal mit dem entsprechendem „Code Smell“ vor dem Refactoring und einmal nach dem Anwenden des Refactorings. „Inverse“ Patterns, die ihre Wirkung gegenseitig aufheben, z.B. „*Pull Up Method*“ und „*Push Down Method*“, wurden jeweils nur einmal untersucht. Außerdem wurden fortgeschrittene Enterprise-Patterns, wie z.B. das Pattern „*Wrap entities with session*“[2], nicht näher betrachtet.

[2] http://www.refactoring.com/catalog/wrapEntitiesWithSession.html (letzter Zugriff 31.08.2012)

Tabelle 3. Übersicht der genutzten Metriken

1) Halstead-Metriken [Ha77]	*Basismesswerte: unterschiedliche Operatoren (n1) und Operanden (n2), Anzahl Operatoren (N1) und Operanden (N2)* Metriken: Vocabulary (Voc), Volume (Vol), Difficulty (Dif), Effort (Eff), Length (Len)
2) LOC- Metriken	Lines of Code (LOC), Commented Lines of Code (CLOC), Statements (Stat), Declarations (Decl).
3) Komplexitätsmetriken	Maximale zyklomatische Komplexität (Max.CC) [MC76], Durchschnittliche zyklomatische Komplexität (CC) [MC76], Maximales Nesting (Max.Nest.)
4) objektorientierte Metriken [CK94]	Lack of Cohesion (LCOM), Depth of Inheritance Tree (DIT), Count of Base Classes (CBC), Count of Coupled Classes (CBO), Count of Derived Classes (NOC), Count of All Methods (RFC), Count of Instance Methods (NIM), Count of Instance Variables (NIV), Count of Methods (WMC)

Insgesamt wurden im Zuge dieser Untersuchung Ergebnisse von 21 bekannten und häufig genutzten Metriken, sowie den 4 Halstead-Basismesswerten (s. Tabelle 3), die zur Berechnung der eigentlichen Halstead-Metriken benötigt werden, in Fowlers 50 Beispielprogrammen erhoben. Entsprechend wurde 1250 Mal der ursprüngliche Code mit der restrukturierten Version verglichen. Ein positiver Wert (> 0) zeigt einen Anstieg der Metrikergebnisse nach dem Refactoring, was bei den allen verwendeten Metriken eine Verschlechterung der Qualität indiziert (vgl. [Ha77, Mc76, CK94]. Ist das Ergebnis negativ, so hat sich der Metrikmesswert verringert. Nach dem Refactoring waren 38% (469) der Metriken höher als zuvor, nur bei 13% (163) der Messungen ergaben sich Verbesserungen, während die restlichen 49% (609) unverändert blieben. Die folgende Tabelle 4 fasst die ermittelten Ergebnisse auf einen Blick zusammen.

Tabelle 4. Übersicht der Veränderung bei den Patterns.

Verschlechterung	469	38%
Unverändert	618	49%
Verbesserung	163	13%
Gesamt	1250	100%

Tabelle 5 und Tabelle 6 schlüsseln diese Ergebnisse für die einzelnen Metriken auf und verdeutlichen, dass bei 14 von 21 Metriken die überwiegende Anzahl von Refactorings zu einer Verschlechterung führt, während nur bei vier die Mehrzahl bei „unverändert“ liegt. Keines der angewandten Refactorings führt zu einer überwiegenden Verbesserung der Metrikwerte und damit zu einer Verringerung der dadurch implizierten Komplexität.

Tabelle 5. Veränderungen klassischer Metriken bei den 50 „Fowler-Refactorings“.

	Len	**Voc**	**Vol**	**Max CC**	**Eff**	**Dif**	**LOC**	**Decl**	**Stat**	**CLOC**	**CC**	**Max Nest**
Verschlecht.	27	28	29	2	29	29	31	26	27	4	0	0
Unverändert	8	11	6	44	5	10	10	21	13	44	50	50
Verbesserung	15	11	15	4	16	11	9	3	10	2	0	0

Tabelle 6. Veränderungen der OO-Metrikwerte bei 50 Refactorings.

	LCOM	**DIT**	**CBC**	**CBO**	**NOC**	**RFC**	**NIM**	**NIV**	**WMC**
Verschlecht.	17	12	12	12	7	21	18	9	19
Unverändert	31	36	36	36	41	25	27	37	27
Verbesserung	1	1	1	1	1	3	4	3	3

Refactorings, die das Volumen des Sourcecodes vergrößern, z. B. durch Hinzufügen von zusätzlichen Klassen oder Methoden, haben per definitionem einen negativen Einfluss auf Halstead- und LOC-Metriken. Zu diesen Refactorings zählen unter anderem *Extract Method*, *Extract Class* oder *Extract Interface*. Entsprechende Ergebnisse werden in Tabelle 7 an Hand konkreter Messergebnisse illustriert. Die Werte zeigen die Veränderung pro Metrik vor einem Refactoring im Vergleich zu einer Messung danach. Beispielsweise hat sich bei *Extract Class* das Halstead-Volume um 30 Metrikpunkte erhöht. Negative Werte in der Tabelle beschreiben eine Verbesserung, bei einem positiven Wert erhöht sich entsprechend die von den Metriken bewertete Codekomplexität.

Tabelle 7. Auszug aus den Differenzen der Halstead-Messungen für verschiedene Refactorings.

Pattern Name	***n1***	***n2***	***N1***	***N2***	**Len**	**Voc**	**Vol**	**Dif.**	**Eff**
Add Parameter	*0*	*2*	*0*	*2*	2	2	7	0	7
Extract Class	*5*	*3*	*6*	*4*	10	8	30	3	90
Extract Interface	*8*	*2*	*8*	*2*	10	10	22	4	44
Extract Method	*5*	*4*	*6*	*5*	11	9	35	3	176
Extract Subclass	*11*	*3*	*12*	*3*	15	14	42	5	108
Extract Superclass	*11*	*3*	*12*	*3*	15	14	42	5	108

Bei „Add Parameter“ wird beispielsweise ein Übergabeparameter zur Methode hinzugefügt. Der Parameter und sein Typ werden in Understand als zwei neue Operanden gezählt und erhöhen somit n2 und N2 und die davon abhängenden Metriken entsprechend.

Die in diesem Kapitel vorgestellten Ergebnisse geben zwar einen ersten Einblick in die Auswirkungen von Refactorings auf Softwaremetriken, sind aber natürlich nicht repräsentativ für die Verteilung der Refactorings, da sie von einer Gleichverteilung in der Praxis ausgehen. Um praxisnähere Werte für die erhobenen Metriken zu erhalten, werden im folgenden Kapitel weitere Ergebnisse aus der Untersuchung zweier exemplarisch ausgewählter Systeme vorgestellt.

4. Fallstudien

Als Untersuchungsobjekte dienten das „Video Store Example“ (ca. 110 bis 230 LOC) aus Fowlers Buch als ein einfaches, aber klar nachvollziehbares „Spielzeugbeispiel“ und das Open-Source-Projekt Lucene (über 140 KLOC) als ein reales, weltweit im Einsatz befindliches System. Wie zuvor wurden die Metriken jeweils vor und nach einem Refactoring erfasst und miteinander verglichen.

4.1 Analyse von Fowlers „Video Store“

Mit Fowlers „Video Store“ [Fo99] wurde zunächst ein beispielhaftes System untersucht, das sieben klar definierte und gut nachvollziehbare Refactoring-Iterationen durchlaufen hat und dadurch, intuitiv nachvollziehbar, besser verständlich geworden ist. Die von Fowler durchgeführten Refactorings wurden Schritt für Schritt nachvollzogen und nach jedem Schritt vermessen. Dazu wurden immer zwei Messreihen durchgeführt, nämlich einerseits der Vergleich der Revision vor dem Refactoring mit der Revision danach und andererseits die Veränderung zwischen der jeweiligen Revision und der ursprünglichen Version (vgl. Tabelle 9). Insgesamt wurden somit 2 Mal in jedem Refactoring-Schritt 21 Metriken und die 4 Halstead-Basiswerte (vgl. Tabelle 3) vermessen, was 350 Messwerte ergab. Im Ergebnis vergrößerten sich bei 57% aller Messungen die Metrikwerte, während sie sich bei nur 9% verringerten. Die entsprechenden Veränderungen sind in Tabelle 8 zusammengefasst.

Tabelle 8. Übersicht Messergebnisse für Fowlers Video-Store.

	LOC	**Halstead Basis**	**Halstead-Metriken**	**OO**	**Gesamt**	**%**
Verschlechterung	50	45	49	56	200	57%
Unverändert	41	4	4	69	118	34%
Verbesserung	7	7	17	1	32	9%
Gesamt	98	56	70	126	350	100%

Besonders interessant sind die Auswirkungen des finalen Schritts, in welchem mehrere Unterklassen eingeführt werden, um die verschiedenen Filmgenres voneinander zu trennen. Dadurch wurden vor allem bei den Halstead-Metriken deutliche Zuwachsraten von teilweise über 100% im Vergleich zur Ursprungsversion (Tabelle 9) gemessen.

Tabelle 9. Halstead-Metriken des finalen Video-Store im Vergleich zur ursprünglichen Version.

Length	**Vocabulary**	**Volume**	**Difficulty**	**Effort**
+77%	+121%	+63%	+110%	+18%

4.2 Analyse des Open-Source-Projekts Lucene

Um die bisher gefunden Resultate an Hand eines realen System zu überprüfen und einen Eindruck über die realen Auswirkungen von Refactorings zu erhalten, wurde abschließend das Open-Source-Projekt Apache Lucene vermessen. Dafür wurden zwölf verschiedene Revisionen aus dem SVN-Repository des Projekts ausgewählt, die ausschließlich mit dem Kommentar „Refactored“ oder „Refactoring“ gekennzeichnet sind. D.h. es wurden – sofern die Kommentare im SVN korrekt gesetzt worden sind – nur Revisionen verwendet, bei denen ein reines Refactoring und keinerlei Veränderung der Funktionalität stattgefunden hat. Für die Messungen wurden jeweils die mit „refactored“ gekennzeichnete Revision und die vorhergehende Revision aus dem Repository geladen; für diese Messungen wurden wiederum die in Tabelle 3 genannten 21 Metriken und 4 Basiswerte erhoben, was für insgesamt 300 Vergleiche ergab: auch bei Lucene ließ sich nur in 21% der Fälle ein verbesserter Messwert nachweisen, eine Verschlechterung aber in 54%; die folgende Tabelle 10 fasst die Ergebnisse zusammen.

Tabelle 10. Übersicht Metrikveränderungen nach Refactoring bei Apache Lucene.

	Absolut	**Relativ**
Verschlechterung	170	57%
Unverändert	63	21%
Verbesserung	67	22%
Gesamt	300	100%

In der folgenden Tabelle 11 wurden die verschiedenen Metrik-Arten in fünf Blöcke unterteilt: der erste Block beinhaltet alle LOC-Metriken, der zweite Block alle Halstead-Metriken, der dritte alle OO-Metriken und der letzte Block alle Komplexitätswerte nach McCabe (vgl. wiederum Tabelle 3). Auch hier zeigt sich deutlich, dass in jedem Block die Anzahl der Ergebnisse, die sich nach dem Refactoring verschlechtert haben, am größten ist. Besonders eklatant sind die Werte für die objektorientierten Metriken, bei denen es nur in acht Prozent aller untersuchten Fälle zu einer Verbesserung kommt.

Tabelle 11. Übersicht der Verteilung nach Metrik-Arten.

	LOC		**Halstead Basis**		**Halstead Metriken**		**OO-Metriken**		**McCabe**	
Verschlechterung	35	58%	32	67%	37	62%	53	49%	13	54%
Unverändert	9	15%	2	4%	0	0%	46	43%	6	25%
Verbesserung	16	27%	14	29%	23	38%	9	8%	5	21%
Gesamt	60	100%	48	100%	60	100%	108	100%	24	100%

Um die Auswirkungen eines einzelnen Refactoring-Schritts besser zu illustrieren, soll an dieser Stelle beispielhaft die Revision 604870 von Apache Lucene genauer betrachtet werden. In dieser wurden im Vergleich zur vorherigen Revision zwei von 7188 Dateien verändert sowie drei neue Dateien hinzugefügt. Laut Revisionskommentar diente das Refactoring dazu, eine gemeinsame Helfer-Klasse zu schaffen ("Refactored to have a common PayloadHelper class"). Von den 21 gemessenen Metriken und 4 Basiswerten steigen 22 zum Teil deutlich an. Zum Beispiel wächst die Anzahl der LOC in den zwei veränderten Dateien um 22%, die durchschnittliche CC um 50%. Nur die Ergebnisse von Halsteads Effort und Lack of Cohesion (LCOM) fallen.

4.3 Fortgeschrittene Qualitätsmodelle

In der Vergangenheit wurde verschiedentlich daran gearbeitet, mehrere Metriken in einen Zusammenhang zu bringen, um aussagekräftige Qualitätsmodelle bzw. -indikatoren für Software zu entwickeln. Ein Beispiel ist der sogenannte Maintainability Index (MI, [Co94]). Dieser basiert auf einem einfachen Stufenmodell, bei dem alle Funktionen über einem bestimmten Schwellwert als qualitativ hochwertig angesehen werden. In den vorgestellten Fallstudien lag keine der vermessenen Methoden unter diesem Wert und auch alle durchgeführten Refactorings ergaben keinerlei Veränderung der Einstufung. Ähnliche Ergebnisse liefert das Modell der Software Improvement Group (SIG) [AYV10] aus den Niederlanden. Auch dort wurden über 90% der Methoden bereits vor dem Refactoring als sehr gut klassifiziert und keine der Einstufungen veränderte sich durch die Refactorings.

In eine ähnliche Richtung gehen auch die Ergebnisse des USUS-Modells [PR10], das in der Lage ist, mehrere Revisionen einer Software miteinander zu vergleichen und sich daraus ergebende Trends zu bestimmen. USUS erkannte nach den Refactorings in jedem der drei vermessenen Projekte einen Trend hin zu schlechterer Qualität und zudem zwei neue Problembereiche (sog. Hotspots) bei den Fowler- Refactorings sowie einen bei Lucene. Tabelle 12 illustriert die Ergebnisse der USUS-Analyse, neben der Veränderung der Hotsports (HS) zeigt sie die Veränderung des Durchschnitts der genutzten Metrik (D) und jeweils die Elemente, die der genutzten Metrik zugrunde liegen.

Tabelle 12. Übersicht der USUS Ergebnisse, negative Trends sind kursiv dargestellt.

		Fowler Patterns		**Video Store**		**Lucene**	
Metrik	**Element**	**D**	**HS**	**D**	**HS**	**D**	**HS**
Average component dependency	Klasse	+0,1	0	*-28,1*	*0*	0	0
Class size	Klasse	0	0	0	0	*0*	*0*
Cyclomatic Complexity	Methode	*-0,4*	*0*	-4,4	0	*0*	*0*
Method length	Methode	0	0	-2,3	0	*0*	*+1*
No. non-static/final public fields	Klasse	*+1,4*	*+2*	0	0	*0*	*0*
Packages w. cyclic dependencies	Package	0	0	0	0	0	0

Auch dieses Qualitätsmodell zeigt also meist nur marginale Veränderungen nach den Refactorings, die zudem oftmals noch negativ interpretiert werden und ist damit nicht in der Lage, die Auswirkungen der Refactorings sinnvoll zu interpretieren.

5. Diskussion

Im vorherigen Kapitel wurden drei aufeinander aufbauende Messreihen mit insgesamt 1900 Vergleichen der bekanntesten Softwaremetriken aus der Literatur vorgestellt. Deren Resultate ähneln sich bei allen drei vermessenen Beispielen – trotz ihrer Verschiedenheit – sehr stark: zumeist steigt die Mehrzahl (insgesamt 47%) der Metrik-Ergebnisse nach einem Refactoring deutlich an, nur bei 15% aller durchgeführten Refactorings verringert sie sich, was auf Grund des großen Umfangs von möglichen Kombinationen stichprobenartig auch analytisch an Hand des Aufbaus der Refactorings und Metriken nachvollzogen werden konnte. Die Ergebnisse sind in Tabelle 13 nochmals auf einen Blick zusammengefasst und zeigen zudem eine große Ähnlichkeit mit den weniger umfangreichen Ergebnissen früherer Untersuchungen ([Du06, SS07]).

Tabelle 13. Übersicht der Auswirkungen aller Messungen im Vergleich zu früheren Veröffentlichungen.

	Ergebnisse dieses Beitrags		[SS07]		[DM03]	
Verschlechterung	983	52%	30	59%	10	50%
Unverändert	848	45%	0	0%	4	20%
Verbesserung	293	15%	21	41%	6	30%
Gesamt	1900	100%	51	100%	20	100%

Erstmals stellt sich mit der in diesem Beitrag durchgeführten Untersuchung heraus, dass Veränderungen nach Refactorings weitgehend unabhängig von der Art der Metrik zu sein scheinen, sowohl die älteren Halstead-Metriken (Verschlechterung von 61%), die bis dato im Zusammenhang mit Refactorings noch nicht untersucht worden waren, als auch jüngere OO-Metriken (Verschlechterung von 56%) verzeichnen deutliche Komplexitätsanstiege. Gleiches gilt auch für die Art des durchgeführten Refactorings, während bisherige Untersuchungen nur einige wenige Refactorings untersucht haben, konnte im Rahmen dieses Beitrags gezeigt werden, dass die 50 bekanntesten Fowler-Refactorings zumeist ähnliche Auswirkungen auf die gemessenen Metriken haben. Es gab allerdings keine Metrik, die sich bei allen Refactorings durchgängig gleich verhalten hätte, so dass belastbare statistische Aussagen noch nicht getroffen werden können.

Interessant sind auch die Erkenntnisse aus der Anwendung fortgeschrittener Qualitätsmodelle, die die Einflüsse der Refactorings sowohl in den einfachen Demonstrationsbeispielen, als auch in einem realen System, weitgehend überhaupt nicht abbilden können. Andererseits verdeutlicht die durchgeführte Analyse des Lucene-Frameworks, wie selbst kleine Änderungen oft großen Einfluss auf Metrikwerte haben können: Im Beispiel aus Abschnitt 4.2 wurden nur insgesamt fünf von ca. 18.000 Dateien durch Refactoring verändert, was nichtsdestotrotz einen merklichen Einfluss auf alle Metriken hatte (s. ebendort). Da alle durchgeführten Messungen nur öffentlich verfügbaren Code verwendet haben, können sie aber jederzeit gut nachvollzogen und auch analysiert werden. Ferner wurde auch für alle Messungen das gleiche Analysewerkzeug („Understand“) verwendet, so dass Verfälschungen der Messergebnisse durch unterschiedliche Metrikinterpretationen ausgeschlossen werden können.

Auf Grund des noch immer recht geringen Umfangs der vorgestellten Untersuchung erscheint eine Verallgemeinerung der Ergebnisse zwar nach wie vor nicht angeraten, es liegen nun aber deutliche Verdachtsmomente vor, dass Softwaremetriken und Refactorings eine gegensätzliche Interpretation von Codequalität aufzuweisen scheinen, so dass weitergehende Forschung in diesem Bereich angeraten erscheint.

6. Fazit und weitere Schritte

Auch nach mehreren Jahrzehnten intensiver Forschung fehlt es den aktuell gängigen Softwaremetriken offensichtlich noch deutlich an „Ausdrucksstärke“, da sie nicht einmal in der Lage scheinen, die Komplexität eines Sourcecodes korrekt zu erfassen, also inhärente Domänenkomplexität von schlechter Programmierung zu unterscheiden. Die in jüngster Zeit populär gewordenen Refactorings zielen genau auf die Behebung schlechter Programmierpraktiken ab und bieten sich entsprechend als eine einfache Möglichkeit für einen Vorher-Nachher-Vergleich von Softwaremetriken an. Die wenigen, bisher veröffentlichten Untersuchungen dieser Art begründeten jedoch den Verdacht, dass gängige Metriken nicht hinreichend in der Lage sind, die vermutlich positiven Einflüsse von Refactorings auf die Quelltextqualität abzubilden. Sie müssen allerdings sowohl im Hinblick auf die verwendeten Metriken, als auch auf die untersuchten Refactorings als wenig umfassend bezeichnet werden. Sie werden von den in diesem Beitrag vorgestellten Untersuchungen um ein Vielfaches übertroffen: neben einer systematischen Vermessung der Fowler-Refactorings wurden hierfür erstmals mehrere Überarbeitungen eines großen Open-Source-System mit gängigen Softwaremetriken untersucht. Die gefundenen Ergebnisse erhärten somit den Verdacht, dass die Metriken nicht in der Lage sind, die durch die Refactorings erreichte Komplexitätsreduktion nachzuvollziehen, wo sie überhaupt eine Veränderung erkennen, ist es meist eine Zunahme der Komplexität. Aus Sicht der Softwaremetriken lässt sich natürlich umgekehrt die Vermutung aufstellen, dass Refactorings die Codequalität negativ beeinflussen könnten; welche Perspektive Recht behalten wird, ist bis dato noch nicht abschließend zu klären. Die Autoren planen daher, in naher Zukunft weitere Open-Source-Projekte zu untersuchen, um die hier vorgestellten Ergebnisse weiter zu untermauern (oder ggf. zu widerlegen). Als zentrales Ergebnis dieses Beitrags bleibt letztlich festzuhalten, dass gängige Softwaremetriken und Refactorings ein weitgehend gegensätzliches „Verständnis" von Codekomplexität aufzuweisen scheinen.

Bei einer stichprobenartigen Betrachtung des Aufbaus einiger Metriken ist dieser Widerspruch zwar nachvollziehbar, genau verstanden sind die Einflüsse von Refactoring auf Metriken aber bei weitem noch nicht. Eine detaillierte analytische Untersuchung der Auswirkungen von Refactorings auf Softwaremetriken ist daher für die Zukunft sicher wünschenswert, aber mit einem nicht unerheblichem Aufwand verbunden. Allein die Kombination der in diesem Beitrag verwendeten 21 Metriken und 4 Basiswerten mit 50 Refactorings ergibt 1250 zu betrachtende Kombinationen, so dass mittelfristig eine weitgehend automatisierbare Quellcode-Vermessung als die ökonomischere Analysemöglichkeit erscheint. Dennoch wollen die Autoren ausgewählte Refactoring-Patterns zukünftig genauer untersuchen, um besser verstehen zu können, welchen Einfluss sie auf den Sourcecode und daraus abgeleitete Metriken haben. Eventuell lassen sich so einzelne Metriken, die Verbesserungen im Code durch Refactorings recht zuverlässig erkennen können, identifizieren und zu einem verbesserten Qualitätsmodell zusammenfassen.

Eine weitere naheliegende Aufgabe ist es, die Auswirkungen anderer Ansätze, die ebenfalls die Codequalität erhöhen sollen, wie z.B. Design oder Architectural Patterns oder auch Coding Guidelines [SA05], auf Softwaremetriken in ähnlicher Weise zu untersuchen. Aus den insgesamt gewonnenen Erkenntnissen lassen sich letztlich eventuell neuartige Metriken (z.B. basierend auf einer Zählung von Code Smells oder Patterns) mit besserer Aussagekraft ableiten oder möglicherweise sogar die erkannten Effekte aus bestehenden Metriken herauszurechnen, um deren Aussagekraft zu verbessern. Der Verdacht, dass heute verfügbare Softwaremetriken und Qualitätsmodelle anerkannte Praktiken guten Softwaredesigns (d.h. Refactorings und Design Patterns, vgl. [Se07]) sowie guter Software-Architektur nicht erfassen können, dürfte jedenfalls immer schwerer von der Hand zu weisen sein.

Literaturverzeichnis

[Al09] Alshayeb, M.: Empirical investigation of refactoring effect on software quality, Elsevier Information and software technology, Vol. 51, Nr. 9, S.1319-1326, 2009

[Al01] Alshayeb, M.; Li, W; Graves, S.; An empirical study of refactoring, new design, and error-fix efforts in extreme programming, 5th World Multiconference on Systemics, Cybernetics and Informatics, 2001

[AYV10] Alves, T.L.; Ypma, C. Visser, J.; Deriving metric thresholds from benchmark data, 2010 IEEE International Conference on Software Maintenance (ICSM), 2010

[BH12] Burger, S.; Hummel, O: Applying Maintainability Oriented Software Metrics to Cabin Software of a Commercial Airliner, 16th European Conference on Software Maintenance and Reengineering, Szeged, 2012

[BS98] Binkley, A.B.; Schach, S.R.: Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures, Intern. Conf. on Software Eng., 1998

[Co94] Coleman, D.; Ash, D.; Lowther, B.; Oman, P: Using metrics to evaluate software system maintainability Computer, IEEE, Vol., 27, S. 44-49, 1994

[CK94] Chidamber S.R.; Kemerer C.K.: A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol. 20. Nr. 6, 1994

[DDV04]Du Bois, B., Demeyer, S., Verelst, J; Refactoring-improving coupling and cohesion of existing code; Working Conference on Reverse Engineering, 2004

[DM03] Du Bois, B., Mens, T: Describing the impact of refactoring on internal program quality, Intern. Workshop on Evolution of Large-scale Industrial Software Applications, 2003

[Ha77] Halstead. M.H: Elements of Software Science (Operating and Programming Systems Series), Elsevier Science Inc., 1977

[Ho08] Hoffmann, D. W: Software-Qualität, Springer-Verlag New York Inc, 2008

[Ka09] Kaur, K.; Minhas, K., Mehan, N., Kakkar, N: Static and Dynamic Complexity Analysis of Software Metrics, Engineering and Technology, Citeseer, Vol. 56, 2009

[JC94] Jones, C: Software metrics: Good, bad and missing Computer, IEEE, Vol. 27, 1994

[Fo99] Fowler, M: Refactoring: improving the design of existing code, Addison-Wesley, 1999

[Fo12] Fowler, M: http://www.refactoring.com/, zuletzt besucht am 21.08.2012

[IO11] ISO/IEC: ISO/IEC 25010: 2011, Systems and software engineering-Systems and software Quality Requirements and Evaluation (SQuaRE)-System and software quality models, Switzerland, International Organization For Standardization, 2011

[KWR10]Khamis, N.; Witte, R.; Rilling, J.: Automatic quality assessment of source code comments: the JavadocMiner, Springer Natural Language Processing and Information Systems, S. 68-79, 2010

[Mc76] McCabe, T.J: A complexity measure, IEEE Transactions on Software Engineering, Vol. 2, Nr. 4, S. 308, 1976

[MR07] van der Meulen, M.J.P.; Revilla, M.A.: Correlations between internal software metrics and software dependability in a large population of small C/C++ programs, International Symposium on Software Reliability, 2007

[MT04] Mens, T.; Tourwé, T.: A survey of software refactoring, IEEE Transactions on Software Engineering, Vol. 30, Nr. 2, S. 126-139, 2004

[NBZ06] Nagappan, N.; Ball, T.; Zeller, A.: Mining metrics to predict component failures, International Conference on Software Engineering, 2006

[Op92] Opdyke W.F.; Refactoring Object-Oriented Frameworks, PhD Thesis, University of Illinois, 1992

[PR10] Philipp, M.; Rauch, N.; Einsicht und Handeln mit Usus, Eclipse Magazin 6.10, 2010

[SA05] Sutter, H.; Alexandrescu, A.:C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Addison-Wesley Professional, 2005

[St02] Stamelos, I.; Angelis, L.; Oikonomou, A.; Bleris, G. L: Code quality analysis in open source software development, Information Systems Journal, Blackwell Science Ltd, Vol. 12, S. 43-60, 2002

[Se07] Seng, O.; Suchbasierte Strukturverbesserung objektorientierter Systeme, Univ.-Verlag Karlsruhe, 2007

[SS07] Stroggylos, K.; Spinellis, D: Refactoring--Does It Improve Software Quality? Fifth International Workshop on Software Quality, 2007

[SZT06] Schröter A.; Zimmermann, T.; Zeller, A.: Predicting component failures at design time. ACM/IEEE International symposium on empirical software engineering, 2006

[WKK07]Wilking, D.; Khan, U.; Kowalewski, S.: An Empirical Evaluation of Refactoring, e-Informatica Software Engineering Journal, Vol. 1, Nr. 1. S. 27-42, 2007

ReActor: A notation for the specification of actor systems and its semantics[*]

Rodger Burmeister
Software Engineering Group
Berlin Institute of Technology, Germany
rodger.burmeister@tu-berlin.de

Abstract: With the increasing use of the actor model in concurrent programming there is also an increased demand in precise design notations. Precise notations enable software engineers to rigorously specify and validate the non-deterministic behavior of concurrent systems. Traditional design notations are either imperative, too concrete, or do not support the actor model. In this paper, we present a new, TLA-inspired specification language called ReActor that supports a declarative style of specification and selected programming language features in combination. For ReActor a precise operational semantics is defined in terms of action interleavings. We propose ReActor to be used in abstract design specifications and as a supplement to existing design notations, especially if a sound notion of concurrent objects is required.

1 Introduction

The execution environments of software have changed significantly in the last two decades. There are trends towards parallel hardware designs, that could not longer be ignored by general software developers. We observe the shift from single processor cores with higher clock rates to many processor cores. Also, computers get continuously connected by high-bandwidth, low-latency networks, which enable access to remote resources as if they where local. Both trends – many-core and distributed architectures – require programming abstractions that can handle concurrent resources in a scalable and maintainable way.

One such abstraction is the so called *actor model* [AMST97, AH87]. Actors are stateful, concurrent objects that use asynchronous, non-blocking *message passing* for inter-object communication. In contrast to threads there is no shared memory and no error-prone mechanisms for synchronizing memory access. Each actor has its own exclusive state space that either is represented by variables (e.g. Scala) or tail recursive function parameters (e.g. Erlang). Variables and parameters can reference other actors. These references can change over time and lead to a dynamic object topology. Actors can activate other actors by sending them messages. Each actor evaluates its messages sequentially, one at a time. Evaluating a message can lead to a local state change, the creation of new actors, further messages, or to an actor's end of life.

[*]Preparation of this work was supported in part by German Research Foundation (DFG) grant HE 6088/1-1.

Messages can be transmitted and evaluated (by individual actors) in parallel. The concrete order in which messages are evaluated may vary for each system run and lead to different behaviors and results. Race conditions and deadlocks may occur in some, rare message order combinations. Testing all potential message order combinations is usually impossible for real world systems due to state space complexity and difficulties in process orchestration. A far better and viable approach would be to specify the actor processes abstractly and validate their combined behavior with the help of tool-supported formal methods, either in the design or implementation phase of the software cycle.

While the actor model is basically the same in all actor languages there are differences in the way messages are handled. Formal notations like Rebeca [Sir06], Temporal Actor Logic [Sch01], Algebra of Actors [GZ01], or Simple Actor Language [AT04, AMST97, Agh86] model pending messages as sets and allow them to be processed in any order. Programming languages like Erlang [Arm07] and Scala [HO07] use queues and simplify the implementation of sequential protocols by features that we call *selective receive* and *relative message order preservation*. The former provides a receiving actor with the ability to define guards for picking a matching message from the pending ones. The latter feature ensures that messages sent from one actor to another actor are transmitted (but not necessarily evaluated) in their sending order.

Formal actor notations with a generic actor model can be used to specify and verify actor models and their properties. Programming features like *selective receive* and *relative message order preservation* are usually not supported but simplify the implementation of sequential protocols. Without these features additional acknowledge-messages are necessary to establish a defined message order. Adding *selective receive* and *relative message order preservation* to formal specification languages would not only improve the readability of a specification but also slim the interfaces of the actors, as these additional messages can be omitted. Another shortcoming of existing actor notations is the imperative style of description. Messages are usually described as concrete sequences of statements. In early phases of software design such implementations are not available. Describing messages in a more abstract way would leave implementation details open to the programmer and later design phases.

In our previous work, we presented a concurrent version of the observer pattern and an implementation of the actor model in TLA$^+$. The effect of messages was modeled abstractly by using pre- and post-conditions. We used the model checker TLC to verify essential safety and liveness properties [BH12] for a limited number of actor-objects. While our model was well-suited to express the pattern's logic it was hard to keep it separate from the underlying actor model as TLA$^+$ has no explicit support for objects or message passing. In this paper, we therefore came up with a special purpose specification language, called ReActor, that hides many details of the actor model in its semantics. Figure 1 illustrates the artifacts and steps of our framework. In this paper, we focus on Step 1, the interpretation and the semantics of ReActor specifications.

In ReActor, we describe each class of objects by an individual specification module. Details like object management and message passing are embedded into the semantics and do not interfere with an application's logic. Both *selective receive* and *relative message order preservation* are supported by ReActor. Actions are described abstractly by pre- and post-

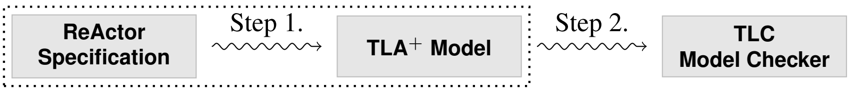


Figure 1: In Step 1, we translate an ReActor-system-specification into a TLA$^+$ model. In Step 2, we use automated model checking techniques to verify safety and liveness properties of concurrent object-based systems. In this paper, we focus on Step 1 and here especially on ReActor's semantics.

conditions. The impact of a message is called *effect*. An operational small step semantics describes how actor objects, messages, and effects globally blend together. We (re)use Lamport's syntax of the Temporal Logic of Actions (TLA) [Lam02] for our declarative expressions in ReActor and for the definition of its semantics.

We assume the reader has a basic knowledge in reading operational semantics [Plo04] and TLA$^+$ specifications [Lam02]. Section 2 introduces ReActor by a small example. An abstract syntax and structure for actors, ReActor specifications, and actor state transition systems is presented in Section 3. In Section 4, we define the semantics of ReActor models by defining their system states inductively.

2 A blinking light example specified in ReActor

ReActor is a declarative specification language for describing actors and their behavior. In this section the notation and its structure is explained by a small example. A ReActor system specification, also called *model*, consists of several *actor modules*, each describing a class of actor objects. An actor module consists of declarations and different kinds of specification schemata. Declarations define constant and variable symbols which can be used in local expressions to refer to an actor's local state. *Schemata* are visual templates in the style of TLA$^+$ and Object-Z [RD00]. They are used to specify interface operations, internal actions, initial conditions, state invariants, and temporal properties. At the expression level ReActor uses a subset of the notation and syntax of TLA$^+$ enriched by some actor-specific elements.

In Figure 2, we present a specification for an actor system that models the behavior of some lights. The system specification consists of two kinds of actor modules one specifying a blinking light and the other the environment. A blinking light actor represents a light that can be turned on and off by sending a $Switch$ message. An environment actor models the user of several blinking lights. It creates and releases lights and switches their status.

The blinking light module declares one local state variable, an initial state predicate and two interface operations. The initial state predicate defines that a light is initially either enabled or disabled. *Initialization schemata* are state-level predicates[1] that logically combine constants, constant operators, and unprimed variables.

Operations like $Switch$ or $Close$ relate incoming message events to local behavior. An *operation* is a transition-level predicate that logically relates constants, constant operators,

[1]TLA$^+$ differs between constant-, state-, transition- and temporal-level expressions.

```
┌──────────── ACTOR Environment ────────────┐
  VARIABLE lights
├─────────────────── INIT ──────────────────┤
  $lights = \{\}$
├────────── INTERNAL AddLight ──────────────┤
  $\exists$ addr $\in$ NEWADDR :
       $\wedge$ CREATE BlinkingLight@addr
            WITH enabled $\mapsto$ FALSE
       $\wedge$ $lights' = lights \cup \{addr\}$
├────────── INTERNAL DropLight ─────────────┤
  $\exists$ addr $\in$ lights : $\wedge$ SEND Close TO addr
                       $\wedge$ $lights' = lights \setminus \{addr\}$
├────────── INTERNAL Switch ────────────────┤
  $\wedge$ $\forall$ addr $\in$ lights : SEND Switch TO addr
  $\wedge$ UNCHANGED lights
└───────────────────────────────────────────┘

┌─ ACTOR BlinkingLight ─┐
  VARIABLE enabled
├──────── INIT ─────────┤
  $\wedge$ enabled $\in$ BOOLEAN
├── OPERATION Switch ───┤
  $\wedge$ $enabled' = \neg enabled$
├── OPERATION Close ────┤
  $\wedge$ TERMINATE
└───────────────────────┘
```

Figure 2: A simple ReActor model with an environment actor that creates, releases and switches lights (on the left) and a blinking light actor that either toggles its status or depletes on external requests (on the right). Vertically aligned conjunctions remove the need for nested parentheses.

primed and unprimed variables, and actor-specific expressions. The *Switch* operation negates a light's status. The *Close* operation defines that an actor has reached its end of life and that it is eventually released. An actor's end of life is claimed by using the actor-specific keyword TERMINATE.

The environment module declares the state variable *lights*, an initialization schema, and three internal actions. The variable *lights* holds references to attached instances of blinking lights. Initially it has to be empty. The set is populated and depopulated by the internal actions *AddLight* and *DropLight*. *Internal actions* in difference to operations do not rely on external message events but on local state preconditions only. We use them to stimulate the system initially, or to model events from the environment, or to describe internal actor transitions. The internal action *AddLight* specifies that a new blinking light is created and that its reference is added to the set of attached lights. Actors are created by choosing a free address ($addr \in$ NEWADDR) and binding it to a concrete instance definition. An instance is defined and bound by a CREATE expression. It takes an actor type (the name of an actor module), an address, and an initial state definition (the equivalent to a constructor) as parameters.

The internal action *DropLight* removes one of the attached lights and sends it a *Close* message. The dispatch of messages is defined by using the SEND expression. A SEND expression requires a message (optionally with parameters) and the receiver's address as input and defines that a message is eventually delivered to a receiver's inbox. Messages sent to unbound addresses will be dropped eventually. The universal quantifier in the *Switch* action is used to send a switch request to all attached lights.

In this small example, we focused on modeling behavioral aspects in ReActor. Beside operations and internal actions ReActor also supports the specification of safety and liveness requirements.

3 Abstract syntax of actors, ReActor specifications, and actor systems

In this section, we present an abstract syntax of actors, ReActor specifications, and actor systems. For each, we define an abstract data type in TLA$^+$. We need these types to introduce ReActor specifications and their semantics in Section 4. In the following, we assume constant sets for messages, actor identifiers (addresses), actor class identifiers, values (assignable to variables and constants), and symbols for naming variables and constants:

$$\text{CONSTANTS } \mathit{Message}, \mathit{ActorID}, \mathit{ClassID}, \mathit{Value}, \mathit{Symbol}$$

The concrete type of these sets depend on the specific ReActor model. For our blinking light example from Section 2 the symbol type is $\{\text{“}\mathit{enabled}\text{”}, \text{“}\mathit{lights}\text{”}\}$. Both variable names from both the actor modules are included in this global type. The value type of $\mathit{enabled}$ is BOOLEAN, because a light can either be enabled or disabled. The value type of lights is SUBSET $\mathit{ActorID}$, because the environment references a variable set of blinking light objects. The overall value type is therefore the union of both these types. In the same way, we can derive the other constants from our specification.

3.1 Actor instances

Actors are unique, stateful objects that react on incoming message events. We model actor instances as TLA$^+$ records with a globally unique address aid, a class identifier cid, a local state lst, and an inbox mbx:

$$\mathit{Actor} \triangleq [\mathit{aid} : \mathit{ActorID}, \mathit{cid} : \mathit{ClassID}, \mathit{lst} : \mathit{LocalState}, \mathit{mbx} : \mathit{Inbox}]$$

A *class identifier* relates an actor instance to a concrete actor module. An instance of a blinking light for example always references the blinking light module. The *local state* is a function that maps constant and variable symbols to concrete values. We assume Value to be a constant set of all assignable entities including references to other actors:

$$\mathit{LocalState} \triangleq [\mathit{Symbol} \rightarrow \mathit{Value}]$$

For a blinking light the value of the variable $\mathit{enabled}$ is either TRUE or FALSE. In ReActor actors are able to process their messages in their relative sending order. Therefore, the inbox is modeled as a sequence of messages instead of a commonly used set:

$$\mathit{Inbox} \triangleq \mathit{Seq}(\mathit{Message})$$

A typical inbox for a blinking light actor may look like $\langle \text{“}\mathit{Switch}\text{”}, \text{“}\mathit{Close}\text{”} \rangle$. This sequence of input messages leads to first switching and thereafter releasing the light. We assume that messages with different parameters are modeled as individual entities of the message type. In our small example messages do not have any parameters.

3.2 ReActor specifications

In ReActor a *system specification* consists of set of class descriptions, which we call *actor modules*. In our example, we defined two such modules one for the environment and another for actors of the type $BlinkingLight$. A actor module defines the state and behavior for a concrete type of actor. It first declares a set of symbols for variables and constants, and then defines different kinds of properties and actions, each represented by its own schema:

$$\begin{aligned} ActorModule \triangleq\ [&cid : ClassID,\ decl : \text{SUBSET}\ Symbol,\ ini : StatePredicate,\\ &int : InternalAction,\ op : Operation] \end{aligned}$$

The blinking light module of our example consists of a variable declaration, an initialization predicate, and two operations. Each actor module relates to a globally unique class identifier cid. This identifier is used to relate an actor instance to a concrete specification module. Declared symbols can be used within any local schema to reference to the variables' or constants' values.

An initialization predicate constrains the type of initial states by some state predicate. A *state predicate* maps the universe of local states to their validity values. In our example, an environment configuration is mapped to TRUE only, if the variable $lights$ is empty. Using such predicate functions enables us to abstract any kind of predicate expressions in our abstract syntax:

$$StatePredicate \triangleq [LocalState \rightarrow \text{BOOLEAN}]$$

A predicate's local state function must include all combinations of declared symbols and value assignments. In our environment example, where $lights$ can reference an arbitrary number of blinking lights, this may lead to infinite domain types. *Internal actions* and *operations* are predicates that define if a local state transition is valid or not. Internal actions rely on an actor's local state only, while operations require an inbox message, too. Both define the validity for the universe of all potential pre- and post-state combinations. A valid transition is permitted by the specification:

$$\begin{aligned} InternalAction &\triangleq [LocalStep \rightarrow \text{BOOLEAN}]\\ Operation &\triangleq [InboxStep \rightarrow \text{BOOLEAN}] \end{aligned}$$

In our example, we used both kinds of transition predicates. In the environment, we have internal actions like $AddLight$ that only rely on the precondition but not on incoming messages. We call this kind of transitions *local step*. In contrast, a blinking light operation like $Switch$ can happen only if the precondition is fullfilled and if there is a corresponding message in the actor's inbox. This kind of transition we call *inbox step*. Both kinds of steps relate a pair of succeeding actor states (pre and $post$) to their transition's effect:

$$\begin{aligned} LocalStep &\triangleq [pre : LocalState,\ post : LocalState,\ eff : Effect]\\ InboxStep &\triangleq [pre : LocalState,\ msg : Message,\ post : LocalState,\ eff : Effect] \end{aligned}$$

An *effect* defines the actors created during a step, the messages sent during a step, and if a step has lead to an actor's end of life. All actors that are created during a step must be bound to a free actor identifier to preserve the uniqueness of actor-identifier relation:

$$Effect \triangleq [out : Outbox,\ new : \text{SUBSET}\ Actor,\ eol : \text{BOOLEAN}]$$

The only effect of our environment's action $AddLight$ is that a new blinking light actor is created: $[new \mapsto \{[cid \mapsto \text{“}BlinkingLight\text{”}, mbx \mapsto \langle\rangle, \ldots]\}, \ldots]$. In ReActor operations and internal actions can claim to send several messages to a target. For example, we could have specified an environment action $SwitchAndClose$ that sends a switch and a close message to each attached blinking light in the same step. Messages that are sent in the same step and to the same receiver can be received in any order. For the $SwitchAndClose$ action this implies that a receiving blinking light can either process the close or the switch message first, which of course is no useful behavior. However, in ReActor we model such unordered sets of messages as a bag for each potential receiver:

$$Outbox \triangleq [ActorID \rightarrow Bag(Message)]$$

If the relative order between messages is important then these messages must be sent in different steps, as we do for $Switch$ and $DropLight$ in our environment specification. For these actions ReActor preserves the relative order of outgoing messages, as they are sent in different steps.

3.3 Actor systems and their semantics

After defining the abstract syntax of actors and actor modules, we now need to relate them. For this, we introduce an abstract representation of an actor system. An *actor system* describes all semantical states and state transitions in terms of a concrete model. It consists of a set of valid system states sst and of a set of actor modules mod:

$$ActorSystem \triangleq [sst : \text{SUBSET}\ SystemState,\ mod : \text{SUBSET}\ ActorModule]$$

The actor modules together define a *model*. The internal actions and operations of that model applied to an initial set of actor instances define how a system can evolve. Our blinking light example always starts with a well initialized environment actor and continues in accordance to the actors' transition predicates. Each transition predicate relates succeeding system states and defines proper system steps. We will use them in the next section to derive target systems states from preceding ones. A *system state* defines the state of the overall system at a point in time. It consists of a configuration of instantiated actors and a configuration of pending messages:

$$SystemState \triangleq [act : \text{SUBSET}\ Actor,\ buf : MessageBuffer]$$

For our blinking light system the system state includes the states of all potential environment and blinking light configurations and all configurations of messages in transit. Messages in transit are messages that are sent but not delivered to the receivers' inboxes yet. They are stored in a global *message buffer*. This buffer is part of the semantical state of a system and consists of a message queue for each pair of sender and receiver:

$$MessageBuffer \triangleq [ActorID \times ActorID \rightarrow Seq(Bag(Message))]$$

Each queue contains a series of message bags. Bags encapsulate a number of unordered messages sent within a single local or inbox step. Succeeding bags define messages sent in different steps. An environment E that has just sent an switch message to an attached blinking light L is buffered as $[\langle E, L\rangle \mapsto \langle \ldots, [\text{“}\mathit{Switch}\text{”}] \mapsto 1\rangle, \ldots]$. We are using queues of bags to implement the targeted feature of *message order preservation*. The concurrent delivery and evaluation of messages is part of ReActor's semantics and will be detailed in the next section.

4 Operational semantics of ReActor

In this section, we present the semantics of ReActor in terms of reachable system states. We first summarize important features informally and then present a precise, formal description.

4.1 Informal description

In ReActor actors are stateful concurrent objects. An actor can either process an operation or an internal action but only one at a time. Enabled internal actions and operations are selected and processed non-deterministically. An internal action is *enabled* if its state precondition is fulfilled. An operation is enabled if its state precondition is fulfilled and a corresponding message was send before. Actors with continuously enabled internal actions or operations must eventually be evaluated (fair scheduling). Each message can be evaluated only once. Sent messages are guaranteed to be delivered. Local messages have no special priority and are treated like any other message. The empty reference and the identity of an actor are represented by the keywords NIL and SELF.

Internal actions and operations represent atomic system steps. Atomic system steps can be interleaved in any order as long as the precondition of each step is fullfilled. Internal actions and operations may alter the local state of an actor, create new actors, send messages, or enable termination. Messages can be sent to known actors only. Actors are known if their identifiers were communicated during initialization, as a parameter of a message, or as the result of a locally created actor. In the blinking light example the environment knows the lights it has created, but the lights do not know the environment as they do not keep any reference to it. Termination eventually leads to the release of an actor. Identifiers of released actors can be reused.

Messages sent in a single atomic step (either internal action or operation) can be received in any order. Messages sent in different atomic steps are received in their sending order as long as the sending and receiving actors are the same (*relative message order preservation*). An actor evaluates its enabled operations in correspondence to the receiving order of its pending messages. Messages that do not correspond to an enabled operation stay unchanged (*selective receive*). A blinking light object with an inbox $\langle \text{“}\mathit{Switch}\text{”}, \text{“}\mathit{Close}\text{”}\rangle$ always evaluates its enabled switch operation first. It is enabled because the precondition of the switch operation is always TRUE. Messages with an invalid destination may be dropped at any time. This may happen if a receiving actor was released before an already sent message was transmitted.

A system can start with any number of well initialized actors. An actor is *well initialized* if its initialization predicate is fulfilled. In the blinking light example, we start with a single environment object. The global message buffer always starts to be empty. Internal actions, like the environment's $Switch$ must be used to trigger initial communication.

4.2 Formal description

After giving an informal summary of ReActor's semantical details, we will now define them more formally. For this, we define the set of all valid system states sst for any system model mod inductively. We will start with valid system initializations (induction basis), and derive and add successor states appropriate to the model consecutively (inductive step). We use inference rules to describe the initialization and derive target states. The following rule reads as "P implies that s is a valid system state":

$$\frac{P}{s \in sst}$$

For ReActor there are five such rules: one for deriving initial system states from the model, one for evaluating an enabled internal action, one for evaluating an enabled operation, one for delivering a pending message, and one for dropping a message with an invalid destination. Non-determinism is covered by the fact that more than one rule may be applied in any of the system states. Figure 3 depicts how each rule contributes to the definition of a system's state space.

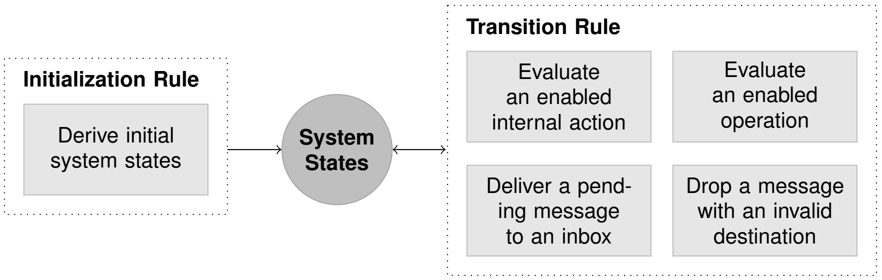

Initialization Rule
Derive initial system states
System States
Transition Rule
Evaluate an enabled internal action
Evaluate an enabled operation
Deliver a pending message to an inbox
Drop a message with an invalid destination

Figure 3: The semantics of a model is defined by first deriving all initial system states and then evaluating a transition rule for each system state inductively.

Each of the rules expresses its premise and implication in TLA$^+$ syntax. All related TLA$^+$ definitions are listed in alphabetical order in Annex A. We used the model checker TLC, type assumptions, and different input values to test all our rules and definitions.

We assume fair scheduling for all transition rules. Actors and transitions that are continuously enabled must eventually be applied. In addition, we assume that model mod is well formed. A model is *well formed* if its data structure and behavioral definitions are consistent. For example the domain of a local state function for a blinking light must match its variable and constant symbols. Due to lack of space, we do not give a full description of well formed models here; but we believe that most of these conditions are obvious by the design of our language.

The first rule – the induction basis – defines the initial system states of a model mod:

$$\frac{\begin{array}{c} s \in \mathit{SystemState},\ s.\mathit{buf} = \mathit{EmptyMessageBuffer}, \\ \mathit{ActorsHaveUniqueIDs}(s.\mathit{act}),\ \forall\ \mathit{act} \in s.\mathit{act} : \mathit{ActorInit}(\mathit{act}, \mathit{mod}) \end{array}}{s \in \mathit{sst}}$$

A system can start with any number of well initialized actors that each must have a globally unique identifier. Our blinking light model always starts with one well initialized environment actor. An actor is well initialized if its state satisfies the actor's initialization predicate and if the actor's mailbox is empty. Also the global message buffer has to be empty. Actors are related to their corresponding actor specifications in the model by using their annotated class identifier. Each system state s that satisfies the rule is a valid starting point for a system run and for our inductive definition of a system's state space. The only valid initialization state in our blinking light example consists of an empty message buffer and an environment actor with no attached lights.

Having defined the set of initial system states, we can now expand it by adding all target states of our atomic system transitions inductively. The first transition rule defines all target states for the evaluation of enabled internal actions:

$$\frac{\begin{array}{c} s \in \mathit{sst},\ a \in s.\mathit{act},\ \mathit{ls} \in \mathit{EnabledLocalSteps}(s, a, \mathit{mod}), \\ s' = \mathit{EvalLocalStep}(s, a, \mathit{ls}) \end{array}}{s' \in \mathit{sst}}$$

An internal action is enabled if there is an actor a that satisfies the internal precondition of the corresponding local step. For the blinking light's environment action *DropLight*, we assume that there is at least one attached light ($\exists\ \mathit{addr} \in \text{NEWADDR} : \ldots$). We define a system's target state by applying an enabled local step and its effect to the corresponding source state. In each system step, we evaluate only one enabled local step at a time (small step semantics). The definition of *EvalLocalStep* implements the evaluation of an enabled local step. It adds newly created actors, enqueues sent messages, releases an actor if it was terminated, and updates the local state accordingly to the local step's specification. For our environment's internal action *AddLight* it redefines a (source) system state by adding a newly created blinking light to the global actor configuration and by updating the environment's local set of lights. Each system state s and succeeding target state s' matching the rule have to be in the system's state space.

The second transition rule defines the target states for the evaluation of enabled operations in the same way we did for the internal actions:

$$\frac{s \in sst,\ a \in s.act,\ is \in \mathit{FirstEnabledInboxStep}(s, a, mod), \\ s' = \mathit{EvalInboxStep}(s, a, is)}{s' \in sst}$$

In contrast to internal actions an operation requires a corresponding inbox message. The message that was received first and corresponds to an enabled operation has the highest application priority (*selective receive*). A blinking light object with $\langle$"Switch", "Close"$\rangle$ as inbox, will always evaluate the switch message first. In the case that the first message cannot be evaluated because of an unfullfilled precondition, the next enabled message gets evaluated. If a operation is evaluated then its corresponding message is removed from the actor's inbox. All transition effects are described by the $\mathit{EvalInboxStep}$ definition of the target state. For a blinking light's Switch operation it removes the first "Switch" message from the actor's inbox and updates the enabled status of that light. All other aspects of the preceding system state stay unchanged.

The rules for evaluating internal actions and operations both add sent messages to the global message buffer. The next rule defines how buffered messages are delivered to their receivers' inboxes:

$$\frac{s \in sst,\ srp \in \mathit{Pending}(s.buf),\ msg \in \text{DOMAIN}\ \mathit{FirstBag}(s.buf, srp), \\ \mathit{Rcv}(srp) \in \mathit{ActorIDs}(sst.act),\ s' = \mathit{EvalTransmission}(s, srp, msg)}{s' \in sst}$$

ReActor delivers only one pending message from any sender to any receiver at a time. In our blinking light example, we deliver either a pending Switch or Close message from the buffer to an blinking light's inbox. The symbol srp represents an arbitrary tuple of sending and receiving actor. The global message buffer stores pending messages for each of these tuples as a sequence of bags. A bag encapsulates messages sent in an internal action or operation step. We transmit the messages of each tuple's front bag first to preserve the order between messages that are sent in different steps. Lets assume $\langle[$"Switch" $\mapsto 1], [$"Close" $\mapsto 1]\rangle$ to be the buffered messages of two environment actions (Switch and Close) for any blinking light, then all messages of the front bag $[$"Switch" $\mapsto 1]$ must be transmitted first. Delivered messages are enqueued to the end of the receiver's inbox and removed from the buffer's message queue. The $\mathit{EvalTransmission}$ definition describes both these effects in terms of a target system state.

The last transition rule defines how to deal with pending messages if the receiver does not exist anymore. For example, the environment action $\mathit{SwitchAndClose}$ from Section 3.2 can lead to situations where a blinking light was already released, but where a Switch message is still pending. Programming languages like Erlang silently drop such messages. In the same manner the sending of messages always succeeds in ReActor even if the receiving actor is not available anymore:

$$\frac{s \in sst,\ srp \in \mathit{Pending}(s.buf),\ msg \in \text{DOMAIN}\ \mathit{FirstBag}(s.buf, srp), \\ \mathit{Rcv}(srp) \notin \mathit{ActorIDs}(sst.act),\ s' = \mathit{EvalDrop}(s, srp, msg)}{s' \in sst}$$

If a pending message targets for a non-existing receiver it is eventually removed from the message buffer without any further effects. Only one message is removed from the global message buffer at a time. The proper processing of sent messages is a property of the model and requires that a receiver stays until all its pending messages were evaluated.

The presented rules together inductively define the set of all reachable system states sst and all valid system state transitions for a given ReActor model mod. All used helper definitions are listed in Annex A.

5 Related work

Our work contributes to the field of actor system specifications and actor semantics. We relate our work to three representative and important works in this fields.

Agha et al. present an actor language and one of the most referenced operational actor semantics [AMST97] in the field. The language describes actors in contrast to ReActor at a lower, more concrete level using an extension of the λ-calculus. External actors and internal receptionists represent interfaces to open environments, while ReActor assumes a fully closed system specification here. Their semantics is defined by a transition relation on actor configurations. Transition steps are defined for each low-level actor operation. We compose the low-level actor operations of each message to a more abstract but semantically also more complex atomic transition step. The message buffer is modeled as a generic bag while we use sequences of bags to support *message order preservation*. Their language and its semantics aims at general application and does not treat with features of popular actor programming languages.

Fredlund et al. present a small step operational semantics for a subset of the programming language Erlang [Fre01] and an improved version that also covers distributed nodes [SF07]. The semantics is separated into two parts, one defines the meaning of functional expressions and the other defines the global weaving of local actor behavior. We use an abstract characterization for our schema expressions and define the global weaving of local actor behavior only. The difference between our abstract characterization and their functional expressions is the granularity the system needs to be modeled in. Our semantics is closely inspired by Erlang and uses the same fundamental abstractions; e.g. mailboxes are modeled as sequences of messages. We add local messages to the global message buffer while Erlang delivers these messages immediately. We do not model dead processes explicitly but remove them instantly. References to acquaintances are defined explicitly by Fredlund and enables reasoning about locality. They implemented their semantics into the model checker McErlang [FS07] that can be used to examine real world Erlang programs and prove liveness and safety properties.

The language Rebeca aims at closing the gap between formal verification and real programs [Sir06]. The core of the language are so called reactive object templates. A template describes the data structure and the behavior of an actor and corresponds to our actor modules. The semantics is defined in an interleaving manner by defining a labeled transition system [SMSdB04]. Controlled behavior for unusual situations is left unspecified, e.g.

for cases where messages should be delivered to invalid destinations. In contrast to our work Rebeca proposes an imperative view to actor operations while we use a declarative approach here. Rebeca describes actor systems in terms of closed system specifications. Open system specifications are supported by components that together implement a closed model. While we use internal actions to specify environmental behavior, Rebeca relies on messages and obligatory initialization actions completely. Different model checker back ends can be used to verify Rebeca implementations.

6 Conclusion and future work

With the trends towards many-core and distributed architectures, engineers also need reliable tools to validate their concurrent software designs. In this paper, we described the formal foundation for an actor-oriented specification and analysis framework. The presented notation is called ReActor and supports a declarative style of description and selected features of actor programming languages in combination. A declarative style of descriptions allows for a more abstract description than in existing actor notations. Implementation details that are unknown in early phases of software design are left open to the programmer and later design phases. Features like *selective receive* and *relative message order preservation* are known from actor programming languages like Erlang and Scala. Both simplify the specification of sequential protocols in asynchronous environments. Supporting them in ReActor helps to slim object interfaces and to improve readability. We detailed the structure of ReActor specifications and presented an operational semantics that defines system states inductively. The semantics of ReActor systems was specified and mechanized in TLA$^+$. Basic assumptions like types were tested using the model checker TLC.

We propose ReActor for specifying actor systems on an abstract but precise level and as a supplement to existing design notations like the UML. Our semantics can be used to strengthen concurrent object notations that lack in a precise semantics for object management or message passing. In the future, we assume an infrastructure that enables the specification of early actor-system designs and the automated validation of safety and liveness properties. Essential features of actor systems are embedded into the semantics of the specification language. This includes the organization of message buffers, the management of object instances, the administration of the address space, and the scheduling of operations and internal actions. With the TLA$^+$-based implementation and validation of an actor-based observer pattern [BH12], we already demonstrated the applicability of our approach.

Overall, we contribute a notation that enables declarative specification of actors and their behavior without fixing implementation details. Future work includes the connection of ReActor to different automatized verification back ends. We also work on better tool support and plan case studies for different kinds of applications.

References

[Agh86] G. Agha. Actors: A Model of Concurrent Computation In Distributed Systems. Technical Report 844, Massachusetts Institute of Technology, 1986.

[AH87] G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, Computer Systems Series, pages 49–74. MIT Press, 1987.

[AMST97] G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[Arm07] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, 2007.

[AT04] G. Agha and P. Thati. An Algebraic Theory of Actors and its Application to a Simple Object-Based Language. In *From Object-Orientation to Formal Methods*, volume 2635 of *LNCS*, pages 26–57. Springer, 2004.

[BH12] R. Burmeister and S. Helke. The Observer Pattern applied to actor systems: A TLA/TLC-based implementation analysis. In *Proc. of the 6th Intern. Conference on Theoretical Aspects of Software Engineering*, pages 193–200. IEEE Press, 2012.

[Fre01] L.A. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology Stockholm, 2001.

[FS07] L.A. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *Proc. of the 12th ACM SIGPLAN Intern. Conference on Functional Programming (ICFP 2007)*, pages 125–136. ACM Press, 2007.

[GZ01] M. Gaspari and G. Zavattaro. An Actor Algebra for Specifying Distributed Systems: the Hurried Philosophers Case Study. In *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *LNCS*, pages 428–444. Springer, 2001.

[HO07] P. Haller and M. Odersky. Actors That Unify Threads and Events. Technical Report LAMP-REPORT-2007-001, Ecole Poytechnique Federale de Lausanne, 2007.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Sofware Engineers*. Addison-Wesley, 2002.

[Plo04] G. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, 2004.

[RD00] G. Rose and R. Duke. *Formal Object Oriented Specification Using Object-Z.* Palgrave Macmillan, 2000.

[Sch01] S. Schacht. Formal Reasoning about Actor Programs Using Temporal Logic. In *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *LNCS*, pages 445–460. Springer, 2001.

[SF07] H. Svensson and L.A. Fredlund. A More Accurate Semantics for Distributed Erlang. In *Proc. of the ACM SIPGLAN 2007 Erlang Workshop*, pages 37–42. ACM Press, 2007.

[Sir06] M. Sirjani. Rebeca: Theory, Applications, and Tools. In *Proc. of the 5th Intern. Conference on Formal Methods for Components and Objects*, volume 4709 of *LNCS*, pages 102–126. Springer, 2006.

[SMSdB04] M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. Modeling and Verification of Reactive Systems using Rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.

A TLA$^+$ definitions

The following definitions complement the rules of ReActor's semantics. We used TLA$^+$ and its base modules (sequences, natural numbers etc.) to define them. The model checker TLC was used to test basic assumptions about these definitions.

```
$ActorIDs(conf) \triangleq \{act.aid : act \in conf\}$

$ActorInit(act, mod) \triangleq$
    $\land\ act.cid \in \{m.cid : m \in mod\}$
    $\land$ LET $schema \triangleq$ CHOOSE $m \in mod : m.cid = act.cid$
      IN $\land$ DOMAIN $act.lst = schema.decl$
         $\land\ schema.ini[act.lst] \equiv$ TRUE
         $\land\ act.mbx = \langle\rangle$

$ActorsHaveUniqueIDs(conf) \triangleq \forall a, b \in conf : (a.aid = b.aid \Rightarrow a = b)$

$BagDecrement(bag, msg) \triangleq$
    IF $bag[msg] = 1$ THEN $[m \in$ (DOMAIN $bag \setminus \{msg\}) \mapsto bag[m]]$
                     ELSE $[bag$ EXCEPT $![msg] = bag[msg] - 1]$

$EmptyMessageBuffer \triangleq [srp \in (ActorID \times ActorID) \mapsto \langle\rangle]$

$EnabledInboxSteps(sst, act, mod) \triangleq$
    LET $schema \triangleq$ CHOOSE $m \in mod : m.cid = act.cid$
    IN  $\{inbox\_step \in$ DOMAIN $schema.op$ :
            $\land\ schema.op[inbox\_step] \equiv$ TRUE
            $\land\ inbox\_step.pre = act.lst$
            $\land\ inbox\_step.msg \in Range(act.mbx)$
            $\land\ ActorIDs(inbox\_step.eff.new) \cap ActorIDs(sst.act) = \{\}\}$

$EnabledLocalSteps(sst, act, mod) \triangleq$
    LET $schema \triangleq$ CHOOSE $m \in mod : m.cid = act.cid$
    IN  $\{local\_step \in$ DOMAIN $schema.int$ :
            $\land\ schema.int[local\_step] \equiv$ TRUE
            $\land\ local\_step.pre = act.lst$
            $\land\ ActorIDs(local\_step.eff.new) \cap ActorIDs(sst.act) = \{\}\}$

$EvalDrop(sst, srp, msg) \triangleq [act \mapsto sst.act, buf \mapsto Receive(sst.buf, srp, msg)]$

$EvalInboxStep(sst, act, istep) \triangleq$
    LET $post\_act \triangleq$ IF $istep.eff.eol$ THEN $\{\}$ ELSE
            $\{[act$ EXCEPT $!.lst = istep.post,$
                          $!.mbx = WithoutFirst(istep.msg, act.mbx)]\}$
    IN  $[act \mapsto (sst.act \setminus \{act\}) \cup post\_act \cup istep.eff.new,$
         $buf \mapsto Send(istep.eff.out, act.aid, sst.buf)]$
```

```
$EvalLocalStep(sst, act, lstep) \triangleq$
    LET $post\_act \triangleq$ IF $lstep.eff.eol$ THEN $\{\}$
                                  ELSE $\{[act$ EXCEPT $!.lst = lstep.post]\}$
    IN  $[act \mapsto (sst.act \setminus \{act\}) \cup post\_act \cup lstep.eff.new,$
         $buf \mapsto Send(lstep.eff.out, act.aid, sst.buf)]$

$EvalTransmission(sst, srp, msg) \triangleq$
    LET $rcv \triangleq$ CHOOSE $a \in sst.act : a.aid = Rcv(srp)$
        $post\_rcv \triangleq [rcv$ EXCEPT $!.mbx = Append(rcv.mbx, msg)]$
    IN  $[act \mapsto (sst.act \setminus \{rcv\}) \cup \{post\_rcv\},$
         $buf \mapsto Receive(sst.buf, srp, msg)]$

$FirstEnabledInboxStep(sst, act, mod) \triangleq$
    LET $enabled\_steps \triangleq EnabledInboxSteps(sst, act, mod)$
    IN  $\{s \in enabled\_steps : \forall\, t \in (enabled\_steps \setminus \{s\}) :$
           $FirstIndexOf(t.msg, act.mbx) > FirstIndexOf(s.msg, act.mbx)\}$

$FirstIndexOf(msg, mbx) \triangleq$
    CHOOSE $n \in$ DOMAIN $mbx$ :
        $\land\ mbx[n] = msg$
        $\land\ \neg\exists\, m \in$ DOMAIN $mbx : (m < n \land mbx[m] = msg)$

$Pending(buf) \triangleq \{srp \in$ DOMAIN $buf : buf[srp] \neq \langle\rangle\}$

$Range(f) \triangleq \{f[x] : x \in$ DOMAIN $f\}$

$Rcv(srp) \triangleq srp[2]$

$Receive(buf, srp, msg) \triangleq$
    LET $post\_bag \triangleq BagDecrement(FirstBag(buf, srp), msg)$
    IN  $[buf$ EXCEPT $![srp] =$ IF $post\_bag \neq EmptyBag$
                                        THEN $\langle post\_bag\rangle \circ Tail(buf[srp]))$
                                        ELSE $Tail(buf[srp])]$

$Send(out, aid, buf) \triangleq$
    $[srp \in$ DOMAIN $buf \mapsto$
        IF $(Snd(srp) = aid) \land (out[Rcv(srp)] \neq EmptyBag)$
            THEN $Append(buf[srp], out[Rcv(srp)])$ ELSE $buf[srp]]$

$Snd(srp) \triangleq srp[1]$

$WithoutFirst(msg, mbx) \triangleq$
    LET $F[seq \in Seq(Range(mbx))] \triangleq$
            IF $Head(seq) = msg$ THEN $Tail(seq)$
                                   ELSE $\langle Head(seq)\rangle \circ F[Tail(mbx)]$
    IN  $F[mbx]$
```

An Adaptive Filter-Framework for the Quality Improvement of Open-Source Software Analysis

Anna Hannemann, Michael Hackstein, Ralf Klamma, Matthias Jarke

Databases and Information Systems
RWTH Aachen University
Ahornstr. 55
D-52056 Aachen, Germany
{hannemann, hackstein, klamma, jarke}@dbis.rwth-aachen.de

Abstract: Knowledge mining in Open-Source Software (OSS) brings a great benefit for software engineering (SE). The researchers discover, investigate, and even simulate the organization of development processes within open-source communities in order to understand the community-oriented organization and to transform its advantages into conventional SE projects. Despite a great number of different studies on OSS data, not much attention has been paid to the data filtering step so far. The noise within uncleaned data can lead to inaccurate conclusions for SE. A special challenge for data cleaning presents the variety of communicational and development infrastructures used by OSS projects. This paper presents an adaptive filter-framework supporting data cleaning and other preprocessing steps. The framework allows to combine filters in arbitrary order, defining which preprocessing steps should be performed. The filter-portfolio can by extended easily. A schema matching in case of cross-project analysis is available. Three filters - spam detection, quotation elimination and core-periphery distinction - were implemented within the filter-framework. In the analysis of three large-scale OSS projects (BioJava, Biopython, BioPerl), the filtering led to a significant data modification and reduction. The results of text mining (sentiment analysis) and social network analysis on uncleaned and cleaned data differ significantly, confirming the importance of the data preprocessing step within OSS empirical studies.

1 Introduction

In the overview article “The Future of Research in Free/Open Source Software Development” (FOSSD) [Sca09] Scacchi identifies a range of research opportunities FOSSD provides for SE. The data collected over years and even decades within communities with up to hundred thousands of participants and large development histories alter fundamentally the costs and calculus of empirical SE. Further, the OSS project management taking place in publicly available Web resources (mailing lists, forums, etc.) provides new opportunities to extend conventional SE cost estimation techniques by rates for social and organizational capabilities. The introduction of new development paradigms like open innovation [Che03] implies integration of enduser communities into SE process. The OSS development presents a successful example of community-oriented organization, which can be used to understand the motivation, role migration, inter-project social networking and community development.

Recognizing the advantages of OSS analyses for SE, a great amount of empirical studies were performed on OSS data in the last decade [GHH$^+$09], [Sca10]. However, the quality of data analysis is directly dependent on the quality of the given data. OSS mailing lists can be messed up by spam. Message content can contain quotations and code clippings. Such "noise" can distort empirical studies executed on OSS data and lead to inaccurate conclusions for SE. During the decade of the knowledge mining in OSS, no general approach for data cleaning has been developed. Till now, some researchers perform data cleaning manually, others apply a combination of some partly automated procedures or even leave out the cleaning step completely. The variety of communication and development infrastructure used by OSS projects poses a special challenge for the automation of data preprocessing.

In this paper, we propose an adaptive and easily extensible Filtering-Framework (FF) for the quality improvement of OSS analysis. The FF also incorporates a schema-matching approach to deal with the variety of communication and development platforms used by different OSS projects. Within the FF, three filters have been implemented: quotation filter, spam filter, and core-periphery filter.

The filters were applied to three large-scale OSS projects from bioinformatics: BioJava[1], Biopython[2] and BioPerl[3]. The filtering led to a significant data modification and reduction. Sentiment and social network analyses were applied to both uncleaned and cleaned data. The analysis results based on uncleaned data differ significantly from those on cleaned data, confirming the importance of the data preprocessing step.

The rest of the paper is organized as follows. The knowledge mining tasks within OSS repositories and the strategies applied for data cleaning within those studies are presented in Section 2. Based on this overview the requirements for data cleaning process are defined. Section 3 describes an approach of our adaptive filtering-framework and its three basic filters for data reduction and content cleaning. The results applying our filtering framework to three OSS projects are shown in Section 4. These results are then discussed and different possibilities for framework extension, depending on a knowledge mining task, are specified in Section 5.

2 State of the Art

In [Sca09], the author presents an overview of opportunities OSS mining brings for SE. In most OSS empirical studies, the researchers use publicly available data within OSS management infrastructures. Presenting the results of a workshop on the future of OSS research held in 2010, Scacchi first describes five sample results on OSS development (OSSD): not only the knowledge extracted from OSS projects, but also the new techniques

[1]http://biojava.org
[2]http://biopython.org
[3]http://bioperl.org

and modeling methods developed for knowledge mining [Sca10]. Defining the research opportunities for OSSD and SE, the author addresses the development of the knowledge mining procedures and systems within and across the OSS projects repositories. But which knowledge mining procedures are relevant for the investigation of development process?

Beside general statistical methods like Lines Of Code (LOC), the procedures use text mining, social network analysis or a combination of both. Already in 2004, Jensen and Scacchi in [JS04] explored the software process discovery in OSS in a more holistic, task-oriented and automated way. They proposed a process model which incorporates text analysis, link analysis, usage and update patterns. They emphasize the need to gather communication data from mailing lists and discussion forums, and not to limit the data source to the code repositories, which only reflect the activities involved in changes to the code. The authors warn against the "well-known problems in text analysis" (e.g. stemming). They also point out that the communication histories "may not accurately reflect" the OSSD and therefore the probabilistic methods are needed to filter the noise from the process instances. For example, the communication can be spread across different resources (e.g. mailing lists and forum); the data given can be outdated (FAQ, wikipage of an OSS project); the communication history can be distorted by spammer messages.

Recently there is a shift from the partly automated analyzing procedures to fully functional frameworks for knowledge mining in OSS [SGK$^+$09], [HCC06]. The Alithea system for OSS analysis [SGK$^+$09] follows a three tier architecture approach: Databases - Cruncher - Presentation Layer. The cruncher calculates several statistical metrics upon (some) database entries and offers the results to the presentation layer. Each metric manages its own database space to store the results and works independent from all other metrics. Whenever a new database entry is added all the metrics are triggered. This approach supports an additive analysis, easy extensibility of the metrics list, metric independence and nesting. However, data preprocessing is not realized within the Alithea system. FLOSSmole[4] [HCC06] addresses the problem of diverse data sources and formats. The project offers a huge data set of diverse OSS projects in multiple formats. The authors also identify the problem that many research teams perform similar analysis tasks on the same data sets and are all struggling with the same problems, for example: where to get necessary data, which datasets to include, or how to implement the analysis (including data cleaning) algorithm.

More and more, the need for automated data cleaning approaches is recognized by the OSS research community. In [BSH09], a study on importance of proper (pre)processing of mailing list data to ensure accurate research results is presented. The authors identify the following challenges for mailing lists mining: message extraction, duplicate removal, language support, attachments, quotes, signatures, thread reconstruction, resolving identity. Since 2010, there is even a workshop series on mining unstructured data with the motto: "Mining Unstructured Data is Like Fishing in Muddy Waters!" [BA10]. However, in most of the empirical studies, the cleaning and analysis is still performed by a bundle of partly automated scripts.

Similar to [MFH02], [BGD$^+$07] apply a set of perl scripts to extract message elements

[4]http://flossmole.org/

and resolve aliasing. However, spam detection within the mailing list is not addressed. Thus, the *indegree* and *outdegree* values calculated in [BGD$^+$07] based on total community size can be distorted, if the community is not cleaned from spammers. In [YNYK04] and [vKSL03], the investigations rely on the user dynamics. In [YNYK04], the issue of data cleaning from spam is not addressed at all, while in [vKSL03], a manual data cleaning from spam is reported. In [JKK11], the authors do extract the spam from the mailing before analyzing "newbies' first interaction". However, the process of spam detection is not described. In [BFHM11], the content-based social network analysis is applied only upon mailing lists of the OSS project R. The authors present a text mining plugin consisting of functions for thread reconstruction, quotation and signature deletion, transformation to British English, synonym replacement, stemming, and stop words. The latter four can be performed either separately or all at once. In another text-mining study of OSS mailing lists [RH07], the authors delete attachments, quoted replies, code and HTML. The transformed results are saved into a separate database. In [VR11], a multi-level ontology for an automated requirements classification within an OSS is designed. The first two levels of the classifier present common natural-language parsing procedure: tokenizer and part-of-speech tagger. Despite the text-mining procedure, the authors do not describe the quotation elimination from the mailing list messages they analyze. From these examples of text-mining, dynamic and social network analysis studies applied to the OSS projects we can conclude that the cleaning steps differ strongly from study to study, introducing variability into data and probably affecting the results. At the same time, different cleaning procedures overlap. Obviously, there is a need for an easily extensible and study-independent filtering-framework.

Specifically, the following requirements can be identified:

Data-Structure Independence: Handling of different data sources and schemata.

Additive Filtering: Possibility to restrict the filtering only to newly added data.

Filter Nesting Possibility to combine arbitrary filters in any order.

Consistent Data Format Consistent structure of results.

Consistent and Easy-to-Use Interface Provide a list of available filters, database instances and required parameters; All filters should be invokable in exactly the same way.

Extensibility: Only the functionality of a new filter has to be implemented, while data exchange between the databases, the framework and the end user should be part of a framework.

Adaptive Database Insertion A parameterized query generator should be included, serving as a mediator between databases and filters.

3 Adaptive-Filtering Approach

OSS projects maintain several management mediums. In order to perform different kinds of analysis upon several OSS projects, there is a need for a filtering approach, which is applicable in the same way on data from any of these mediums. An *artifact* is defined as an elementary contribution in such a management element, like one post in a forum or one mail in a mailing list. The framework can be provided with a set of artifacts that need to be filtered. As these artifacts define different properties, a mapping from semantic meaning – known by the framework – to the corresponding property of the artifact is needed.

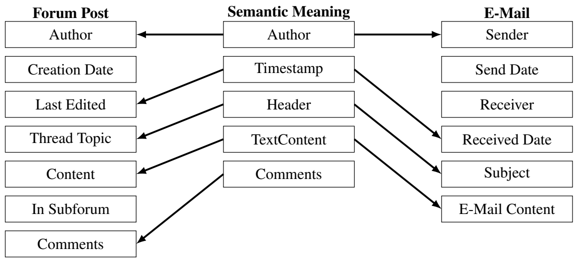


Figure 1: An Exemplary Mapping for Forums and Mailing Lists

Figure 1 gives an exemplary mapping for forums and mailing lists. In order to add a new artifact type or a new project, a database connection and a mapping table have to be specified. Given this information, the filters of the framework can be extended for any additional data sources.

A multi-threading approach, as presented in Figure 2, can speed up the filtering. The concept is to check when the last time this filter was run on the given data source. If there was no data update in the meanwhile, then the results of the last filtering are returned. If there was a data update after the last filtering, the system can clean the new data either in a synchronous or in an asynchronous way.

In the asynchronous way, the already filtered data subset is directly returned to the requester so that the next analysis task can be started immediately upon the pre-filtered data. In parallel, the filtering process upon the non-filtered data is applied. If the data was not previously filtered at all, the requester will get the data after filtering in subsets of a predefined size. With this technique, the requester accesses the first results very quickly and can already start analyzing preprocessed artifacts, while further filtering is continued in the background.

In the synchronous way, all new artifacts are filtered first, and then the whole data set is returned to the requester. This alternative is applied, if the requester needs all artifacts at once for further analysis/filtering (for example social network analysis), but it significantly increases the requesters waiting time for the results.

This approach is quite different to the one presented for Alithea [SGK$^+$09], where the computation starts on each update. In our case, the filtering is triggered on demand, as a large total amount of filters is expected, but only a small subset is needed for each analysis

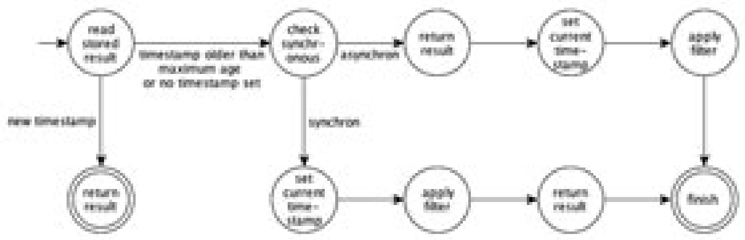


Figure 2: Finite State Machine to Receive Filtered Data

task. Additionally, filters may be designed for a special type of input data (e.g. user-list vs. mailing-list). Hence, computing a filter for user lists on a mailing list would create unnecessary overhead. A further argument for this approach is that in many cases, only certain data subsets are significant for the analysis. These subsets can be created using fast and simple filters on the entire dataset. The more time-consuming filtering can then be applied only to this subset. Again this prevents computing unused overhead. Also filtering on a daily basis is rare in research, mostly it occurs in intervals alternating intensive and no analysis.

Two distinct types of filters can be implemented in our framework: **Dataset Reduction Filters** and **Artifact Transformation Filters**. For performance issues, **Content Cleaning Filters**, a subset of Artifact Transformation Filters, are distinguished in this paper as well. All types return artifacts in a predefined format. In order to construct a complex filter, it is possible to define a sequence of filters $f_1, f_2, \dots f_n$, which is then applied to the data. The results of filter f_1 serve as the data source for f_2, leading to sequences of arbitrary many filters in an arbitrarily order. A user just needs to trigger the last filter of the filter sequence as for each further filter in the sequence the updating process described above is triggered as a cascade. In contrast to Alithea system [SGK+09], which allows only independent metrics, in our framework filters can be implemented which require a pre-filtered data format.

3.1 Dataset Reduction Filters

In order to analyze an OSS project, the data set should be reduced to those artifacts which are significant for the analysis goal. E.g. if the analysis goal is to identify communities of users collaborating on the same issues based on their communication, only discussions should be taken into account. For this issue, Dataset Reduction Filters (DRF) are designed:

$$f_i \in DRF : ARTIFACT \to ARTIFACT, a \mapsto \begin{cases} a, & \text{if conditions are met.} \\ \emptyset, & \text{otherwise.} \end{cases} \tag{1}$$

These filters are used to remove artifacts that do not match certain conditions defined by the filter. Referring to the community identification example before, one of these filters checks if an artifact is part of a discussion, then it is kept, otherwise it is removed. As a reference implementation a spam-filter is presented based on the Bayes Decision Rule (BDR):

$$\text{BDR} : \mathit{WORDS} \rightarrow \mathit{CLASSES}, w_1^N \mapsto \hat{c} = \arg\max_c \{\log(pr(c)) + \sum_{w=1}^{W} N_w \log(p(w|c))\}$$

For this rule a set of N words is needed, delivered by the text content of the artifact.

$$w_1^N := a.\mathit{TextContent}$$

For spam-filtering, only two classes are relevant: SPAM and HAM.

The decision is based on the minimal posterior risk that the set of words w_1^N belongs to class c with the assumption that only the amount of words matter, the ordering is not taken into account. The equation presents a simplified way to decide, if an artifact is most likely *SPAM* or not. But still one problem is left: the exact distributions of all words among the classes is not known. These distributions are derived from manually labeled artifacts, the assumption is that for a random subset the same distribution of words holds as for the complete set. Hence the subset has to be large enough to fulfill this assumption. One special case to be considered is if no maximum can be found: an artifact has equal probabilities for *SPAM* and *HAM*. The behavior is defined via a parameter “onEqual” for each filtering process individually, leading to the following filter function:

$$f : \mathit{ARTIFACT} \rightarrow \mathit{ARTIFACT}, a \mapsto \begin{cases} a, & \text{if BDR}(a) = \mathit{HAM} \\ \emptyset, & \text{if BDR}(a) = \mathit{SPAM} \\ \text{onEqual}(a), & \text{otherwise.} \end{cases}$$

$$\text{where: BDR} : \quad \mathit{ARTIFACT}.\mathit{TextContent} \rightarrow \mathit{CLASSES}, \\ w_1^N \mapsto \hat{c} = \arg\max_c \{\textstyle\sum_{w=1}^{W} N_w \log(p(w|c))\}$$

3.2 Artifact Transformation Filters

For many tasks, preprocessing of the data is necessary, e.g. aliasing of users for mailing-list archives. The same preprocessing is required for several tasks, hence it is useful to store the intermediate result. As in the source data, these intermediate results are not considered, they have to be stored in an additional location. The filter-framework introduces Artifact Transformation Filters (ATF) to provide this functionality:

$$f_i \in \mathit{ATF} : \mathit{ARTIFACT} \rightarrow \mathit{ARTIFACT}, a \mapsto a' \tag{2}$$

These filters are used to modify parts of the artifact. This modification can either be overwriting an attribute of or adding an additional attribute to the artifact.

The results of an ATF are again artifacts, which can be used for further filtering. However, the user has to be careful as the modified attribute may influence the behavior of filters based on it. In order to reduce redundant data storage only the modified attribute is stored by the filter, all other attributes are requested from the original artifact, or earlier filters respectively. An additional benefit of this storage method is that ATFs might be skipped during computation of filter sequences, if the ATF results do not affect the behavior of the requesting filter. For example, a spam filter is not affected by an ATF for aliasing and therefore these could work in parallel and the latter could get ahead of the ATF. As soon as the transformed attribute is requested, the requester has to wait for the ATF to finish.

A core-periphery filter (CPF) was implemented as one example for the ATF filter type adding an additional attribute. This filter iterates over a list of artifacts, most likely users, and takes the network as input, in form of a database table or a filter. At first, CPF computes values for each node in the network based on its out-degree and orders the nodes by this value in ascending order: $\{N_1, N_2, \ldots, N_n\}$ with out-degree(N_i) $\leq$ out-degree(N_{i+1})$\forall i \in \{1, \ldots n-1\}$. Iteratively CPF removes the first node from this ordered list and determines two values:

k_{i-1}: The value CPF assigned in the last iteration, starting with $k_0 = 0$.

o_i: The out-degree of the removed node, considering only edges leading to nodes still in the list.

The highest of these two values is then assigned to the node which was just removed: $k_i = \max(k_{i-1}, o_i)$. Besides that, CPF determines the two succeeding nodes N_j, N_{j+1} with the largest difference between their assigned values k_j and k_{j+1}. After this iteration the nodes N_i are mapped to the artifacts in the dataset. For the successfully mapped nodes the attribute *COREPERIPHERY* is added. The assigned value is either *Core* or *Periphery* based on the node's assigned value k_i:

$$CPF : ARTIFACT \to ARTIFACT, a \mapsto a' \tag{3}$$
$$\text{where: } a'.\text{COREPERIPHERY} = \begin{cases} \text{Core} & \text{if } a \mathrel{\widehat{=}} N_i \wedge k_i \geq k_j \\ \text{Periphery} & \text{if } a \mathrel{\widehat{=}} N_i \wedge k_i < k_j \end{cases}$$

During the evaluation we experienced that the last two nodes in the list get assigned a much higher value than all other nodes. Those nodes represent managers of OSS projects. Therefore we restrict $j < n - 2$, such that these two are not used to determine the largest difference, leading to better results.

3.3 Content Cleaning Filters

The analysis of longer texts is known to be a very time-consuming task. In order to enhance the performance of the analysis algorithm, unnecessary information, e.g. HTML-tags, should be extracted beforehand. Additionally some algorithms need the text in a special format, e.g. "part-of-speech"-tagged, which could also be done beforehand. For these issues, the Content Cleaning Filters (CCF) are designed as a subset of ATFs.

$$f_i \in CCF : ARTIFACT \to ARTIFACT, a \mapsto a' \tag{4}$$

These filters are used to transform the text content of artifacts. This transformation is considered as a special case, as the text content in general consumes much more space than all other attributes. To keep the overhead for the general case as low as possible this transformation is stored in a different table, offering more space for the value.

A quotation filter is presented as a reference implementation for the CCF filter type. Most web-based communication tools offer a possibility to quote other users. For analysis of user communication, these quotations are of high importance. However, for analysis of the content they are superfluous and lead to blurred results, as the same content gets analyzed again every time it is quoted. These quotes most of the times follow a certain pattern depending on the tool used for communication e.g. in many forums:

```
[QUOTE]Some quoted text[/QUOTE]
```

Hence, the quotation filter is applied to search for these patterns. If one or more patterns are found in the text content of an artifact, a copy of the text is generated in which the quoted parts are removed. The cleaned message is stored in the database. This behavior can be mathematically described as follows:

$$QF : WORDS \to WORDS, w \mapsto \begin{cases} \emptyset, & \text{if } w \text{ inside a pattern.} \\ w, & \text{otherwise.} \end{cases}$$

Hence the function for this filter is defined as follows:

$$f : ARTIFACT \to ARTIFACT, a \mapsto a'$$
$$\text{where: } a'.TextContent = QF(a.TextContent)$$

4 Validation in three OSS Projects

To validate our adaptive-filtering approach, it has been applied to the mailing lists of three large-scale OSS projects – BioJava[5], Biopython[6], and BioPerl[7].

Bayesian Spam Filter. In order to evaluate the spam filter, a test set of 1000 mails was extracted randomly out of the overall 60.190 mails. They were manually filtered and labeled. Afterwards the same data subset was automatically filtered, and the results were compared with the manual classification, shown in Table 1.

According to this evaluation result, no ham was wrongly removed by the filter. Around half of the spam mails were automatically identified. Figure 3 shows an excerpt of the spam messages and spammer ratio in the BioJava project from June 2003 to June 2005.

[5]http://biojava.org
[6]http://biopython.org
[7]http://bioperl.org

		Correct	
		Ham	Spam
	in test set	976	24
Selected	Ham	976 (100.00%)	13 (54.17%)
	Spam	0 (0.00%)	11 (45.83%)

Table 1: Bayes Spam Filtering

		Correct	
		Normal	Quotation
	in test set	11662	21646
Selected	Normal	11115 (95.31%)	1365 (6.31%)
	Quotation	547 (4.69%)	20281 (93.69%)

Table 2: Quotation Filtering

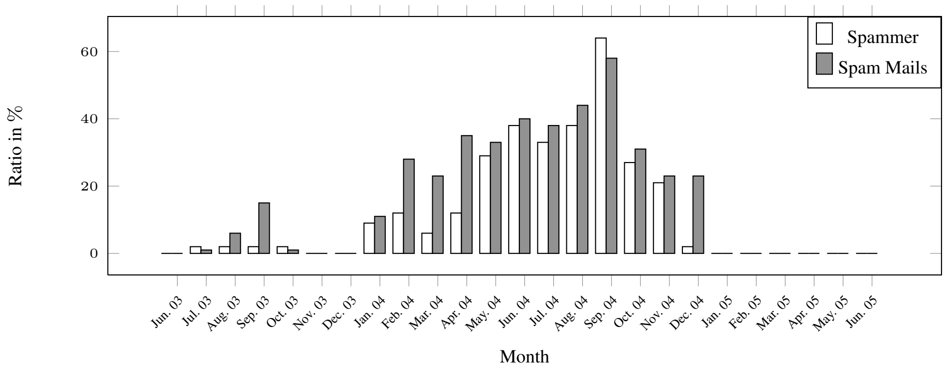


Figure 3: Spam distribution in BioJava Project

Obviously, the amount of spam is neither constant – fluctuating between 0% and 60% – nor insignificant – covering up to 60% of the communication. Similar fluctuations could be measured in all projects. The considerable and non-monotone variation of the spam volume is consistent with previous findings in spam analysis [Faw03].

Quotation Filter The Quotation Filter presented in Section 3 was also applied on these three OSS projects. Again, to evaluate the filter a test set of 100 mails was extracted randomly and the quotations were removed manually. In total, this test set included 33308 words, of which 21646 words were in quoted text, presenting 65% of the original data. This manually filtered set was compared word by word with the results of automatic filtering. Table 2 displays the results of the comparison.

The significant modification of data collected form OSS repositories after filtering gives evidence to the following general conclusions concerning spam within mailing lists: A *non-constant fraction of spammers* over the years was detected in diverse projects. Due to the variable level of spamming activity a *distortion of dynamics* occurs in the uncleaned data, hiding the important changes within the community. In addition, a *non-negligible rise of spammers* up to 60% could be observed. Hence, a major part of uncleaned data is spam not relevant for OSS analysis. Further, the *analysis could be accelerated* by removing spam from the input data and, thus, significantly reducing the data set. Additionally, the *identification of spammers* or other user classes can be applied to investigate their influence on the OSS project success [HKJ12].

Based on the results of the quotation filter we can conclude: A *non-negligible amount of quotations* – about 65% of the content – can be detected in user communication. These quotations form a large amount of text which is contained multiple times in the overall communication. But they do not add any additional content information. Hence, *text-mining* can be significantly *speeded-up* by removing the quotations and thereby reducing the input data. Additionally, *textual classification improvements* could be achieved as the quotations belong to other authors and therefore should not be taken into account.

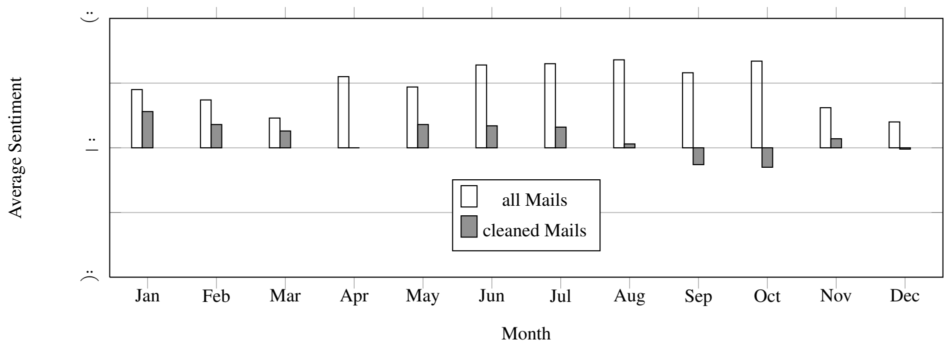


Figure 4: Sentiment Analysis for BioJava in year 2004

Furthermore, the influence of this filtering on OSS analysis was investigated. The mailing lists were cleaned, using the spam and the quotation filter. First, two measures of social network analysis - indegree and density - were applied to uncleaned and cleaned data. In [BGD$^+$07], *indegree* was defined as “the proportion of the total population that has responded to this candidate since his/her first post”. In 2004, 951 different users have posted an email to the BioJava mailing lists. The spam filter identified only 626 non-spammers - one third people less then in the original data. The indegree values calculated using uncleaned data are about one and half times smaller than indegree values calculated using data after filtering. The density, a basic measurement of social network analysis, calculated on the cleaned data is increased by a factor of 2.3.

Next, a sentiment analysis, a text-mining procedure which allows to estimate the sentiment of a message (either positive or negative) [Liu11], was performed on the original and filtered data. The average sentiment of all three OSS projects under study per month was calculated by subtracting the number of negatively labeled mails from the number of positively labeled mails. The values were normalized by the total number of mails per month. Figure 4 shows the sentiment of the BioJava community in 2004. The white bars represent the sentiment estimated on uncleaned data, while the grey bars represent the sentiment estimated on cleaned data. The latter is in the most month less positive. This shift can be explained by the positive nature of spam advertisement messages. Additionally, there is an effect resulting from quotation deletion. It insignificantly influences the average, but shifts many mails from positive to negative sentiment or vice versa. For instance, the following mail[8] is labeled as negative, however, after removing the quotations it is labeled as positive:

[8]http://lists.open-bio.org/pipermail/biojava-l/2004-March/004512.html

```
Maximilian H. wrote:
> Hallo everyone,
> Another question, this time a rather stupid one: How can I get rid of
> those."Originally:."/."Now:.".messages of the demos? [...]
I found some code in my copy in org.biojava.bio.gui.sequence.GuiTools
that is printing these messages (lines .33 and .34). These lines are
commented out in the version in CVS (revision 1.4). [...]
Matthew
```

This effect is due to the fact that someone describes a bug or a problem with negative emotion. The answer contains a solution with positive sentiment, but quotes the negative description as well. The description is often much longer than the solution, hence the mail would be recognized as negative in total. If the problem description is removed, only the solution part is analyzed and gets labelled as positive. The same situation can also occur the other way round: someone announces a nice feature (positive) and gets a bug report back (negative).

5 Conclusion and Outlook

Publicly available OSS communication and development histories provide a range of new opportunities for empirical studies of development process, which are not possible within conventional SE projects. However, the quality of a study is strongly depended on the quality of the data used for analysis. A "noise" within OSS data can lead to inaccurate conclusions for SE. Within this paper, an adaptive filter-framework for the quality improvement of OSS analysis has been presented. The framework uses a semantic mapping approach and a plugin technique for different database systems. Therefore, new data sources can be added quickly. It offers an easily extensible set of elementary filters, to clean and preprocess data sets. By using sequences of these elementary filters more complex filters can be constructed. Any filter sequence can be individually constructed/reused for each data set. Reusability is provided by a strict interface to the filters and a consistent data format within the framework: Each filter can be accessed in the same way, independent of the underlying data set and filtering function. In order to reduce redundant filtering to a minimum, an incremental filtering approach is implemented. Data sources of OSS projects are continuously growing. The approach allows to filter only newly arrived data, which has not been filtered so far. Quotation and spam filters of the framework were validated in three large-scale OSS projects - BioJava, Biopython, and BioPerl. A significant data reduction after both spam and quotation filtering was measured. A non-monoton rise of spammers up to 60% could be observed. The quotations form a large amount of text - about 65% - within OSS mails. Thus, the cleaned data set is much smaller than the original one. The results of text mining and social network analysis on the uncleaned OSS data significantly differ from the results of the same procedures on the cleaned data. The importance of data preprocessing step for OSS analysis was confirmed.

In future, we plan to enhance the presented framework by further frequently required filters. Plenty of studies apply text mining for OSS analysis. Despite different goals of the studies, in most cases, several preliminary preprocessing steps of textual data are similar. For example, part-of-speech tagging can be realized as a new filter. However, OSS communication brings some new challenges to part-of-speech tagging: First, users often posts source code snippets, which do not follow a natural language and therefore cannot be tagged correctly. Next, many artifacts contain informal language and domain-related abbreviations, which are also hard to tag correctly with a training set in formal language. To handle each of these two challenges, a separate filter can be designed.

References

[BA10] Bettenburg N.; Adams B.: Workshop on Mining Unstructured Data (MUD) because "Mining Unstructured Data is Like Fishing in Muddy Waters"! In Reverse Engineering (WCRE), 2010 17th Working Conference on, pages 277 –278, oct. 2010.

[BFHM11] Bohn A.; Feinerer I.; Hornik K.; Mair P.: Content-Based Social Network Analysis of Mailing Lists. The R Journal, 3(1):11–18, June 2011.

[BGD$^+$07] Bird C.; Gourley A.; Devanbu P.; Swaminathan A.; Hsu G.: Open Borders? Immigration in Open Source Projects. In Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07, pages 6–, Washington, DC, USA, 2007. IEEE Computer Society.

[BSH09] Bettenburg N.; Shihab E.; Hassan A.: An empirical study on the risks of using off-the-shelf techniques for processing mailing list data. In Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, pages 539 –542, sept. 2009.

[Che03] Chesbrough H. W.: Open Innovation: The new imperative for creating and profiting from technology. Boston: Harvard Business School Press, 2003.

[Faw03] Fawcett T.: "In vivo" spam filtering: a challenge problem for KDD. SIGKDD Explor. Newsl., 5(2):140–148, December 2003.

[GHH$^+$09] Godfrey M. W.; Hassan A. E.; Herbsleb J.; Murphy G. C.; Robillard M.; Devanbu P.; Mockus A.; Perry D. E.; Notkin D.: Future of Mining Software Archives: A Roundtable. IEEE Softw., 26(1):67–70, January 2009.

[HCC06] Howison J.; Conklin M.; Crowston K.: FLOSSmole: A collaborative repository for FLOSS research data and analyses. International Journal of Information Technology and Web Engineering, 1(3):17–26, 07 2006.

[HKJ12] Hannemann A.; Klamma R.; Jarke M.: Soziale Interaktion in OSS. Praxis der Wirtschaftsinformatik, February 2012.

[JKK11] Jensen C.; King S.; Kuechler V.: Joining Free/Open Source Software Communities: An Analysis of Newbies' First Interactions on Project Mailing Lists. In System Sciences (HICSS), 2011 44th Hawaii International Conference on, pages 1 –10, jan. 2011.

[JS04] Jensen C.; Scacchi W.: Data Mining for Software Process Discovery in Open Source Software Development Communities. In Proceedings Workshop on Mining Software Repositories, 2004.

[Liu11] Liu B.: Web Data Mining. Springer, 2 edition, 2011.

[MFH02] Mockus A.; Fielding R. T.; Herbsleb J. D.: Two case studies of open source software development: Apache and Mozilla. ACM Trans. Softw. Eng. Methodol., 11(3):309–346, July 2002.

[RH07] Rigby P. C.; Hassan A. E.: What can OSS mailing lists tell us? A preliminary psychometric text analysis of the Apache developer mailing list. In Fourth International Workshop on Mining of Software Repositories, 2007.

[Sca09] Scacchi W.: Understanding Requirements for Open Source Software. In Lyytinen K.; Loucopoulos P.; Mylopoulos J.; Robinson B., editors, Design Requirements Engineering: A Ten-Year Perspective, pages 467–494. Springer-Verlag, 2009.

[Sca10] Scacchi W.: The Future Research in Free/Open Source Software Development. In Proc. ACM Workshop on the Future of Software Engineering Research (FoSER), pages 315–319, Santa Fe, NM, November 2010.

[SGK$^+$09] Spinellis D.; Gousios G.; Karakoidas V.; Louridas P.; Adams P. J.; Samoladas I.; Stamelos I.: Evaluating the Quality of Open Source Software. Electronic Notes in Theoretical Computer Science, 233:5 – 28, 2009. Proceedings of the International Workshop on Software Quality and Maintainability (SQM 2008).

[vKSL03] von Krogh G.; Spaeth S.; Lakhani K. R.: Community, joining, and specialization in open source software innovation: a case study. Research Policy, 32(7):1217–1241, 7 2003.

[VR11] Vlas R.; Robinson W. N.: A Rule-Based Natural Language Technique for Requirements Discovery and Classification in Open-Source Software Development Projects. In Proceedings of the 44th Hawaii International Conference on System Sciences - 2011, 2011.

[YNYK04] Ye Y.; Nakakoji K.; Yamamoto Y.; Kishida K.: The Co-Evolution of Systems and Communities in Free and Open Source Software Development. In Koch S., editor, Free/Open Source Software Development, pages 59–82. Idea Group Publishing, Hershey, PA, 2004.

On the Prediction of the Mutual Impact of Business Processes and Enterprise Information Systems

Robert Heinrich, Barbara Paech

Institute of Computer Science, University of Heidelberg
Im Neuenheimer Feld 326, 69120 Heidelberg, Germany
{heinrich, paech}@informatik.uni-heidelberg.de

Abstract: Frequently, the development of business processes and enterprise information systems (IT systems) is not well aligned. Missing alignment of business process design and IT system design can result in performance problems at runtime. Simulation is a promising approach to support the alignment of the designs by impact prediction. Based on the predicted impact, the designs can be adapted to enable alignment. However, in current simulation approaches, there is little integration between business processes and IT systems. In this paper, we present a simulation-based approach to predict the impact of a business process design on the performance of IT systems and vice versa. We argue that business process simulation and IT simulation considered in isolation is not an adequate approach as this neglects the mutual impact on workload distribution. Furthermore, we sketch a solution idea to adequately represent workload distribution in simulation.

1. Introduction

In the development of business processes and IT systems, the non-trivial mutual impact between both is often neglected. Performance problems at runtime such as long IT response times, long process execution times, overloaded IT systems or interrupted processes often result from missing alignment of business process design and IT system design. For example, if an IT system is invoked by too many actors in a process, its response time will increase or the IT system might even not be available any longer. As a result, the process is impeded or even interrupted. Simulation is a powerful approach to predict the impact of a business process design on the performance of IT systems and vice versa. Business process design and IT system design can be aligned by making adjustments based on the predicted impact.

Simulation approaches are widely used in the business process domain as well as in the IT domain. In the business process domain, simulation is applied to predict business process performance and financial impact (e.g. [HJK97]). In the IT domain, computer network simulation is used to estimate the performance of network topologies. Moreover, software architecture simulation is applied to evaluate the performance of component-based architectures (e.g. [BKR09]) as well as service-oriented architectures (e.g. [Sav12]). These approaches are adequate to predict the performance of business processes or IT systems in isolation as they are focused on a certain domain. Currently, there is little integration between the business process domain and the IT domain in simulation approaches. However, integration is required to predict the impact of one domain on the other. In literature, there are few approaches (cf. [Pai96], [GPK99], [Bet12]) concerned with the alignment of business processes and IT systems via simulation. These approaches attempt to address missing alignment by exchanging information between isolated simulations but do not consider the mutual impact of business processes and IT systems in detail.

In line with these approaches, we developed a simulation-based approach to predict the impact of a business process design on the performance of supporting IT systems and vice versa. Based on our approach, we discuss how the mutual impact of business processes and IT systems is reflected by isolated simulations and present limitations of the approach. Moreover, we propose the integration of business processes and IT systems within a single simulation as a solution to adequately represent the mutual impact.

Our approach supports several roles in the joint development of business processes and IT systems using model-based performance simulation. Based on the predicted impact:

i. Requirement engineers can verify in the design phase whether an IT requirement can be fulfilled by a proposed IT system design for a given process design.
ii. System developers can compare proposed design alternatives of IT systems invoked in a given process without implementing IT system prototypes.
iii. Hardware administrators can evaluate the utilization of IT resources such as CPU or hard disk drive for a proposed IT system design or process design.
iv. Business analysts can verify in the design phase whether a process requirement can be fulfilled by a proposed process design and a given IT system design.
v. Process designers can compare process design alternatives without performing a process in practice. The IT system impact is included in the comparison.

The paper is structured as follows: In Section 2, we introduce definitions required to understand the paper. We describe the mutual impact of business processes and IT systems in Section 3. In Section 4, we discuss related approaches. Our approach is presented in Section 5. In Section 6, we describe in detail how to derive an IT usage profile from a business process model. In Section 7, we argue why business process simulation and IT simulation considered in isolation is not an adequate approach to represent the mutual impact of business processes and IT systems in simulation. Moreover, we propose an integration of both simulations. Section 8 concludes the paper and presents future work.

2. Definitions

A *business process* is a “set of one or more linked activities which collectively realize a business objective or policy goal, normally within the context of an organizational structure defining functional roles and relationships” [WMC99]. Each activity within the business process is composed of a set of one or more linked steps. Steps either are performed completely by a human actor – called *actor step*s – or performed completely by an IT system – called *system step*s (or system calls). A human actor performs actor steps lined up in his/her *worklist*. A worklist is similar to a waiting queue of an IT resource (cf. [BKR09]) in which jobs to be processed by the IT resource are lined up. Business processes are typically specified on several levels of abstraction which are composed hierarchically. Processes consist of subprocesses, subprocesses are composed of activities, and activities consist of actor steps and system steps. Hierarchical composition is useful to keep track of large processes. We focus on the level of steps as fine-grained actor steps and system steps are needed to predict the mutual impact of business processes and IT systems in simulation. In Section 3, the mutual impact is discussed in detail based on an example.

A *business process instance* is the "representation of a single enactment of a process" [WMC99]. An *IT system instance* is an executable representation of the IT system design. A business process design P and an IT system design S are *aligned*, if

i. System steps of S are invoked in P, and
   for all the process instances P' of P and all the IT system instances S' of S holds:
ii. S' meets the requirements of S when used in P', and
iii. P' meets the requirements of P when uses S'.

Workload is "the amount of work to be done" [Oxf12]. Business process workload determines the amount of process instances that traverse the business process. Often workload is measured in process instances per time unit. Process instances traverse all the actor steps and system steps on a certain path through the process from the process starting point to a process end point. The total time required by a process instance to traverse a system step is called *response time*. The total time required by a process instance to traverse an actor step is called *execution time*. The time needed to traverse an activity within the process or an entire process is named execution time, too. If the workload does not change over time, it is called *time-invariant*. If the workload changes over time, it is called *time-variant*.

The difference in time in which the process instances reach a certain point in the process is called *distance*. The distance in which two process instances start the execution of the process is named *inter-arrival time*. *Workload distribution* refers to the distance between process instances within the process. For example, three process instances can occur with a constant distance (30 seconds) between each other, or they can occur in *burst*s, e.g. all three process instances occur directly after another or even at the same time. In all cases, the workload is three process instances in one minute. While traversing the process, the distance between process instances can vary. Bursts of process instances or gaps between the process instances can emerge.

3. Mutual Impact of Business Processes and IT Systems

In this section, we present a process from practice in order to describe the mutual impact of business processes and IT systems based on an example.

3.1 The Process of Order Picking

Currently, we are conducting a case study to apply the approach presented in the paper in practice. As the case study is not yet completed, we show a simplified representation of the process of the case study as an example in this section.

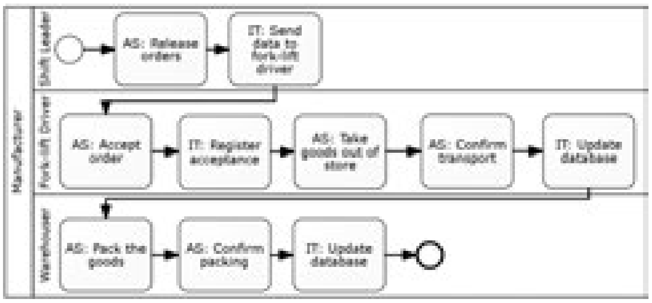


Fig. 1: The Process of Order Picking

Figure 1 shows the process of order picking in the stock of a manufacturer. Goods requested by an order are taken out of the stock and are packed to be transported by trucks. Requested orders have to be packed for transport at the time the trucks arrive. Delays are very expensive in the logistics business. Thus, it is a time-critical process. The process has a starting point and an end point, depicted by circles. Steps are visualized by rectangles with rounded corners. “AS:” denotes actor steps. “IT:” denotes system steps. Arrows visualize the control flow in the process. Lanes represent organizational roles of human actors. For each role, several human actors are available. Thus, orders can be processed concurrently. The shift leader releases the orders that need to be packed. The IT system inserts the order data into a database and transfers the order data from the database to a mobile client of the fork–lift driver. Then, the fork–lift driver accepts the order. The IT system registers the acceptance in the database. The fork–lift driver takes the goods out of the stock. S/he puts the goods to a place where they are packed for transport. Afterwards, s/he confirms the transport. The IT system updates the database and informs the warehouser. The warehouser packs the goods for transport and puts them to a place where they are picked up by trucks later. Then, s/he confirms the transport. Lastly, the IT system updates the database.

3.2 Process Impact on IT System Performance

The IT system performance is affected by the business process design as well as by the business process workload. The business process design specifies for example, which system steps are invoked in the process, when a specific system step is invoked and which system steps are invoked concurrently. Suppose a business process that includes two system steps of an IT system. The IT system performance may differ depending on whether the steps are invoked sequentially or concurrently. The Business process workload represents the amount of process instances that traverse the process, as introduced in Section 2. Process instances traverse all the actor steps and system steps on a certain path through the process. Consequently, the business process workload also determines how often an IT system is invoked. The IT system performance may differ depending on whether the system is invoked once per second or 100 times per second.

3.3 IT System Impact on Process Performance

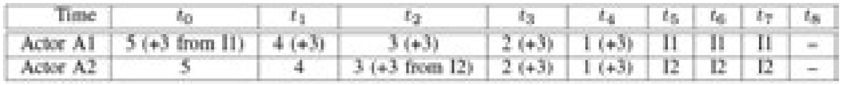
The IT system performance affects the business process performance twofold. First, if the IT system is overloaded as too many actors invoke the system, it is no longer available for actors in the business process. As a consequence, the execution of the business process is impeded or even interrupted. For example, if the order data cannot be transferred to the mobile client of the fork–lift driver (e.g. as the IT system is overloaded by too many actor requests), the goods may not be available for loading the trucks in time. Second, the response time of system steps may affect the business process performance if their extent is comparable to the extent of the execution time of the actor steps within the process. In this case, the IT system response time may significantly increase the execution time of the entire process or of single activities within the process. See Section 2 for a definition of response time and execution time. For example, in our case study, the transfer of order data to the mobile client of the fork–lift driver lasts up to 40 minutes and more which heavily impairs the process execution time as it extends accordingly.

3.4 Mutual Impact of Actor Steps and System Steps on Workload Distribution

The IT system performance as well as the business process performance is affected by workload distribution within the process. A definition of workload distribution is given in Section 2. According to Mi et al. [Mi08], workload distribution has "paramount importance for queuing prediction, both in terms of response time mean and tail" as it impacts performance significantly. Unequal distribution of workload (i.e. burstiness) often leads to increasing IT response times. The more unequal the workload is distributed, the higher can be the variation in the lengths of the waiting queues. Increased queue lengths result in increased waiting times. Human actors as well as IT resources process steps in a similar manner. They both process steps lined up in their work list or respectively their waiting queue at a certain processing rate. Thus, it is straightforward that workload distribution impacts the execution time of actor steps (i.e. the process performance) in the same way as it impacts the response time of system steps (i.e. the IT system performance). The Workload distribution in the process is affected by actor steps as well as by system steps. For example, assume the FIFO (First-in first-out) scheduling principle. If an actor is already busy when an actor step is to be performed by this actor, the execution of the actor step must wait until the actor is ready to perform the actor step. If an IT resource used in a system step is already busy when it is invoked by an actor request, the request must wait until the resource is ready to process the request. Several other scheduling policies are possible for human actors and IT resources. Waiting times hinder the process instances in traversing the business process. For each step in the process, waiting times may differ from process instance to process instance. Consequently, the distances between the process instances in the process may vary during the process execution. Table 1 shows an example of how the distance between process instances is decreased which may then result in a burst.

Assume there are two actors A1 and A2 of the role fork–lift driver. The waiting queues of both actors have a length of five time units at t_0. There are two process instances I1 and I2 which reach the actor step “AS: Accept order” at a distance of two time units. Both instances put a demand of three time units on the actors. At the point in time t_0 the process instance I1 is allocated to the actor A1, because the queues of both actors have the same length but A1 is the first in line. At the point in time t_2 the process instance I2 is allocated to actor A2, because its waiting queue is the shortest, now (see Table 1). The processing of I1 is completed at the point in time t_7. The processing of I2 is completed at the point in time t_7, too. I1 and I2 reach the system step “IT: Register acceptance” at the same time which, according to Mi et al. [Mi08], can result in a higher mean response time than if the process instances reach the system step with a distance of two time units. Note: according to [HJK97] we assume in this example that each actor has the same processing rate. Different processing rates can be taken into account in the future (cf. [HHP12]).

Table 1: Decreasing the Distance



Time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
Actor A1	5 (+3 from I1)	4 (+3)	3 (+3)	2 (+3)	1 (+3)	I1	I1	I1	–
Actor A2	5	4	3 (+3 from I2)	2 (+3)	1 (+3)	I2	I2	I2	–

4. Related Work

We conducted a literature research on approaches that consider business processes as well as IT systems in performance simulation. The literature research included the digital libraries of ACM, IEEE and Springer as well as Google Scholar as sources of publications. There are few related approaches in literature. In analogy to our approach, these approaches use process simulation and IT simulation in isolation as described in the following. In contrast to our approach, related approaches are described on a high level and do not provide details on the representation of the mutual impact in simulation.

The approaches by Serrano and den Hengst [SH05] as well as Tan and Takakuwa [TT07] only predict the impact of IT systems on process performance but do not consider the impact of processes on IT system performance.

We identified three approaches that predict the impact in both directions – process to IT and IT to process. Painter et al. [Pai96] propose a three-layered structure to describe the relationship between business processes, information systems and computer networks. This approach uses business process simulation and computer network simulation in isolation in order to predict process and IT performance. IDEF 3 models that represent information systems are used as a “middle” layer between business processes and computer networks. Giaglis et al. [GPK99] present an approach to support concurrent engineering of business processes and IT and to facilitate investment evaluation. Business process simulation is used to predict business process performance. Computer network simulation is used to depict several alternative network architectures and topologies. IT level analysis provides the link between the business process and computer network levels, although the IT level itself may not be explicitly included in the simulation models. Betz et al. [Bet12] sketch a framework to integrate the lifecycles of business processes and business software for requirements coordination and impact analysis. However, also this approach uses business process simulation and component-based software architecture simulation in isolation.

All these approaches do not clearly define interfaces between business processes and IT systems. Using separate simulations requires the specification of an IT usage profile based on the business process. This is needed in IT simulation to adequately represent the impact of business process design and business process workload on IT system performance as described in Section 3.2. However, none of these approaches describes the specification of an IT usage profile. Performance metrics supported by the approaches are not described. The approaches do not provide enough details to understand how the mutual impact of business process and IT system is represented. Moreover, it is hard to assess their applicability in practice. Thus, we decided to develop our own approach to analyze how isolated simulations represent the mutual impact of business process and IT system. This is described in the following.

5. An Approach to Predict the Mutual Impact

The prediction of the mutual impact of business processes and IT systems requires well defined interfaces between both domains in order to transfer information from one domain to the other. The interfaces are visualized in Figure 2.

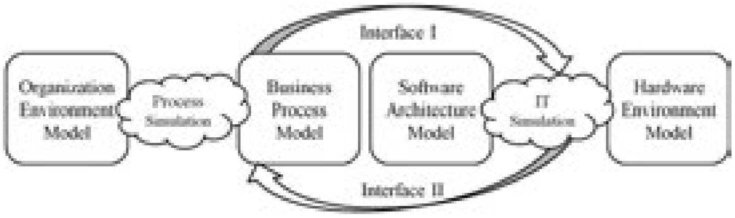

Interface I
Organization Environment Model
Process Simulation
Business Process Model
Software Architecture Model
IT Simulation
Hardware Environment Model
Interface II

Fig. 2: Interfaces between Business Process Simulation and IT Simulation

The central artifact of our approach is the business process model. The business process model depicts the actor steps and system steps as well as their dependencies. It describes the behavior of the human actors involved in the process and the interaction between human actors and the IT systems. The organization environment model contains information about the human actors in the organization such as their roles or availability. The software architecture model includes information about the software components and their dependencies. The hardware environment model contains information about the hardware nodes the software components are allocated to. Examples of information represented in the models and further details on the simulation concepts we build upon can be found in [HJK97] and [BKR09]. From the business process model, an IT usage profile for the IT performance simulation is derived. The results of the IT simulation are written back to the process model to be available for the process performance simulation. Thus, the IT usage profile and the extended business process model characterize the alignment of business processes and IT systems in simulation.

The approach consists of two parts which are used for different purposes. Part 1: The performance of an IT system is predicted based on a specific IT usage profile derived from a business process model. The prediction output can be used by requirement engineers to verify a proposed IT system design against requirements and by system developers to compare alternative IT system designs. Moreover, hardware administrators can check the utilization of hardware nodes for a proposed IT system design or business process design, and adapt the hardware if necessary. Part 2: The performance of a business process is predicted considering the performance of supporting IT systems. The prediction output can be used by business analysts to verify a proposed process design against requirements and by process designers to compare alternative process designs. Figure 3 gives an overview of the approach.

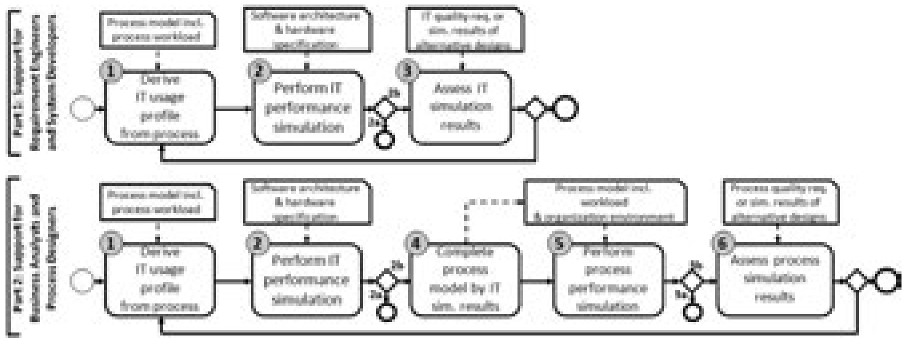

Part 1: Support for Requirement Engineers and System Developers
Process model incl. process workload
Software architecture & hardware specification
IT quality req. or sim. results of alternative designs
1
Derive IT usage profile from process
2
Perform IT performance simulation
2a
2b
3
Assess IT simulation results
Part 2: Support for Business Analysts and Process Designers
Process model incl. process workload
Software architecture & hardware specification
Process model incl. workload & organization environment
Process quality req. or sim. results of alternative designs
1
Derive IT usage profile from process
2
Perform IT performance simulation
2a
2b
4
Complete process model by IT sim. results
5
Perform process performance simulation
5a
5b
6
Assess process simulation results

Fig. 3: Overview of the Approach

The single steps of our approach are described in the following ordered by their number.

Step 1, an IT usage profile, which contains information about the usage of the IT system, is derived from the process model (Interface I in Figure 2). This step consists of three substeps detailed in Section 6.

Step 2, IT performance simulation: The derived IT usage profile and the existing software architecture model and hardware environment model are used for IT performance simulation. The models include performance-specific annotations such as IT resource demands (see [BKR09] for details). Based on queueing networks [Bol98], the IT performance simulation predicts the utilization of IT resources and the response times of the invoked IT system steps. The simulation result indicates one out of two possible cases:

a. The IT system is overloaded for the given usage profile. In this case, a stable mean response time of the system steps cannot be predicted. The procedure ends. There are two ways to handle this case. The first way is to reduce the workload of the business process (if it is acceptable to change this requirement) or to revise the process design. The second way is to revise the IT system design (including software design and hardware design).
b. The IT system is not overloaded. In this case, the simulation predicts the mean response time per system step and the utilization (ratio busy/idle) of the IT resources.

Step 3, the output of the IT performance simulation is compared to IT requirements for requirements verification. For example, for each system step the predicted mean response time is compared pair wise to a mean response time requirement. If the requirement is not met, the IT system design or the process design or the requirement has to be adapted. In the case of design changes, the approach (steps 1-3) has to be repeated. Moreover, simulation output can be used to compare design alternatives based on the predicted performance. Part 1 of our approach is necessary as requirements may be missed on the IT level even if all requirements on the business level are met. For example, an actor has finished the business process in time, but s/he is not satisfied because s/he had to wait several minutes for each system answer.

Step 4, the process model is completed by the IT simulation results (Interface II in Figure 2): Actor steps within the process typically last several minutes. The response time of a system step is often in a millisecond range. In this case, there is no need to consider the response time values of the system steps in process simulation as they have only negligible impact on the process performance. In the case that the response time of a system step is in its extent comparable to the execution time of an actor step, it must be considered in process simulation. For example, large database requests, complex calculations or data transmission to mobile systems as in our case study may have a response time of several minutes. So, the response times are annotated to the corresponding system steps in the process model.

Step 5, process performance simulation: The completed business process model is used for process performance simulation together with information about the organizational environment such as the availability of actors. Based on queueing networks, the process performance simulation predicts the utilization of human actors and the execution times of the actor steps within the process. The response times of the IT system steps, if annotated to the process model, are used as predicted in step 2. The simulation result indicates one out of two possible cases:

a. The actors cannot handle the assigned actor steps as there are too many steps assigned. A stable mean execution time of the actor steps cannot be predicted. The procedure ends. There are two ways to handle this case. To revise the process design (e.g. rearrange the steps or allocate more actors) or to reduce the process workload (e.g. reduce the number of process instances).
b. The actors can handle the assigned steps. The simulation predicts the mean execution time per actor step and the utilization (ratio busy/idle) of the actors. Moreover, it predicts the mean execution time of the entire process and of the single activities within the process.

Step 6, the output of the process performance simulation is compared to the process requirements for requirements verification. If the process requirements are not met, the process design, the IT system design, or the process requirements have to be adapted. In the case of design changes, the approach has to be applied again (steps 1-3 and 4-6) to predict the impact of the changes. Moreover, the simulation output can be used to compare design alternatives based on the predicted performance.

The proposed approach allows verifying requirements against a business process and IT system design. If the requirements are not met by the simulation results, there are basically two options: Adapting the requirements or adapting the design of the process or the IT system respectively. The decision what to adapt is highly dependent on the specific context and requires expertise. So our approach cannot identify what is best to adapt. However, our approach provides guidance. For example, if overloaded resources limit performance, our approach shows which resources are overloaded. Moreover, our approach provides decision support by exploring the impact of several design alternatives using performance simulation. Thus, our approach supports the selection of an adequate design out of a set of proposed design alternatives.

Our approach is not focused on a specific simulation tool support. In order to realize our approach, we build upon the existing tool supports of the simulation approaches ADONIS [HJK97] and Palladio [BKR09]. ADONIS enables the simulation of business processes and organizational resources such as actors. Palladio allows the simulation of software architectures on component level and of IT resources.

6. Deriving an IT Usage Profile from a Business Process Model

In this section, we discuss step 1 of our approach in detail using the example of Palladio by describing three substeps necessary to create an IT usage profile based on a process model. The substeps are not specific to Palladio but generally applicable. In Palladio, the interaction between actors and the IT system is described in the form of one or more usage scenarios contained in a usage model [BKR09]. Each simulation in Palladio refers to one usage model. For each usage scenario, a time-invariant workload has to be specified which represents the load on the IT system. The challenge of step 1 is to create usage scenarios and related workload specifications based on a process specification.

As a first substep 1.1, a business process is segmented into subprocesses so that each subprocess can be traversed by process instances without interruptions and has a time-invariant workload specification. The segmentation highly depends on the specific process. One has to consider when a specific part of the process is performed, which parts of the process are performed concurrently and thus may create concurrent accesses to IT resources (concurrent process parts have to be simulated concurrently), and which parts of the process exclude each other and thus have to be simulated separately (e.g. one part is performed during the day, another part is performed at night). For example, suppose a business process consisting of a sequence of three IT-supported activities A, B and C. Each day 1000 process instances start the execution of the process. In the following, we discuss two different cases.

First case: A is performed during the day (8am to 10pm). B consists of calculations the IT system does exclusively at night (10pm to 8am). C is performed the next day (8am to 10pm). Thus, the process is segmented into three subprocesses. Each subprocess covers one of the activities A, B or C. Two separate IT simulations are required. The first simulation covers A and C as two concurrent subprocesses (the day simulation). Although A and C are performed on different days, except for the first and the last day of process execution, every day A and C are performed concurrently. The workload of each subprocess in this simulation is 1000 process instances in 14 hours or on average one process instance every 50.4 seconds. The second simulation covers the subprocess B (the night simulation). The workload of this simulation is 1000 process instances in 10 hours. The simulation can be split as B is executed during the night and A concurrent to C

(A || C) is executed during the day. B and A || C exclude each other. Thus, there are no concurrent IT resource demands between B and A || C.

Second case: B is not executed at night but also executed during the day but only from 8am to 1pm. Concurrent resource demands are possible. Thus, B is a subprocess concurrent to A || C which has to be considered in a simulation concurrently to A and C. The workload of the subprocess of B is 1000 process instances in 5 hours. The second case has to be investigated using the two separate IT simulations A || B || C from 8am to 1pm and A || C from 1pm to 10pm. There is no activity performed during the night. The two separate simulations may result in different mean response times of the same system steps of A and C. Thus, we recommend using the maximum of the two values per system step for further processing. The result of substep 1.1 is a set of subprocesses which have a time-invariant workload and which can be traversed without interruptions.

In substep 1.2, the workload of each system step in the subprocess is determined based on the workload of the subprocess and the control flow of the subprocess (e.g. probabilities of path branches or number of loop iterations). The determination of the workloads of the single system steps is based on the assumption that there is no overload of the IT resources or the actors (i.e. that continuously process instances arrive at the system step in a certain distance). If the assumption is incorrect, it will become apparent in the course of our investigation (see step 2 in Section 5). Depending on the control flow of the subprocess, the workload of each system step within the subprocess is calculated as demonstrated in Figure 5.

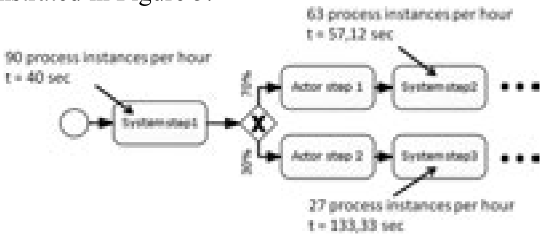


Fig. 4: Determining the Inter-arrival Time

For example, there are three system steps in the subprocess. The subprocess workload is specified in the form of an average inter-arrival time (t) of 40 seconds. Moreover, there is a branch. The probability of one path is 70%. The probability of the other path is 30%. Based on the probability of the path branch, the process instances split up on the paths which results in the inter-arrival times depicted in Figure 5. In the case of parallel steps, there is no distribution. In the case of a loop, the workload adds up depending on the number of loop iterations.

Finally, in substep 1.3, each system step in the subprocess is represented by a usage scenario in a Palladio usage model, which only contains the system step and which is annotated by its specific workload. Moreover, for usage model creation, one has to consider which subprocesses are performed concurrently and which exclude each other as described above. All the system steps of concurrent subprocesses have to be modeled within the same usage model.

7. Discussion of the Approach

In contrast to related approaches, we worked out a detailed description of our approach as presented in Section 5 and 6. This was required to analyze how the mutual impact is represented using business process simulation and IT simulation in isolation. In this section, we discuss how the mutual impact is reflected by our approach and show limitations identified while working out the approach. Moreover, we argue for a new approach that integrates business processes and IT systems in a single simulation.

In our approach, IT system performance is considered as a factor of process performance. The business process model is extended by IT simulation results before process simulation as described in Section 5. Thus, using our approach, IT system performance impacts process performance.

IT simulation requires a specification of the usage of the IT system. Using process simulation and IT simulation in isolation requires deriving an IT usage profile for IT simulation from the business process specification. We described the derivation of an IT usage profile in Section 6. However, the execution of a business process typically lasts several hours or even days. Thus, there may be changes in the workload during the execution, which have to be represented in simulation, too. For example, from 9am to 11am the process is triggered 100 times per minute, and from 1pm to 5pm, the process is triggered 50 times per minute. From 11am to 1pm, the process is not triggered at all, due to a lunch break. As shown in the example, inter-arrival time changes over time. IT simulation approaches typically rely on the strong assumption of time-invariant workloads. Workloads changing over time cannot be mapped to a time-invariant workload without approximation. Approximation usually results in reduced accuracy of the predicted IT response times. We are currently extending Palladio to include time-variant workloads in simulation to address this open issue.

An important finding of the research presented in this paper is that using isolated simulations, workload distribution (e.g. bursts) within the process is not correctly represented in simulations. Workload distribution within the process is only influenced by system steps in IT simulation or by actor steps in process simulation respectively. The impact of steps of the other domain on workload distribution is neglected in both simulations. Thus, in the example in Table 1, the distance between both process instances would not decrease in IT simulation which may result in a distorted response time prediction. Moreover, suppose actors interrupt their work in the lunch break mentioned above. Process instances are stuck in the work lists of the actors and cannot reach the IT resources. As the IT resources keep on processing process instances from their waiting queue during lunch break, waiting queues of IT resources empty during lunch break. This effect cannot be represented in simulation without considering the actor step's impact on workload distribution. As workload distribution may impact the performance significantly, the prediction accuracy of approaches using isolated simulations is limited. Mi et al. [Mi08] showed how big the deviations can be. In an experiment, they observed bursts in workload distribution of an IT system. They compared the response time of a system step in the case of a random workload distribution to the response time of the system step in the case that all the requests are compressed into a single large burst. In case of the burst, they observed that the mean response time is approximately 40 times longer than in random distribution. The 95th percentile of the response times is nearly 80 times longer in bursts. Although this is an extreme example, it demonstrates that one cannot expect accurate simulation results by neglecting workload distribution. As human actors process jobs in a similar manner as IT resources, we expect similar results for actor steps in the case of bursts.

Although the impact of workload distribution on performance is easily comprehensible, to the best of our knowledge the mutual impact of actor steps and system steps on workload distribution was not discussed in literature before. All the related approaches presented in Section 4 do not consider the mutual impact on workload distribution as they use isolated simulations.

Nevertheless, in performance prediction often a rough estimation is sufficient. For example, one wants to check whether a system step lasts a few milliseconds or minutes, or whether a process comes to a standstill because too many steps are assigned to the actors. Therefore, approaches based on isolated simulations are helpful although they have limited prediction accuracy. Thus, our approach is adequate for performance prediction in many cases. For this reason, we currently apply our approach in practice. In those cases where a higher precision is needed, e.g. if there is little distance between requirements and expected performance, another approach is required. We propose the integration of business processes and IT systems within a single simulation as a solution to adequately represent the mutual impact of actor steps and system steps on workload distribution. An integrated simulation seems to be a promising approach, as there are several analogies if we abstract from the different semantics of business process simulation and IT simulation. Both kinds of simulations...

- ... can be built upon queuing networks (queuing theory [Bol98]).
- ... simulate the utilization of resources (human actor resources or IT resources). A human actor resource is the representation of a human actor in simulation.
- ... use a specification of a workflow of actions to be processed by the resources, either in the form of a process model or in the form of an IT usage profile.
- ... use actions that can be composed hierarchically.
- ... use a specification of workload.
- ... acquire and release shared passive resources. Passive resources in a business process are non-IT devices or machines such as a fork-lift. Passive resources in IT systems for example are threads in a thread pool or database connections. They are available in a limited capacity and shared among all process instances.

Palladio seems to be an adequate foundation to be extended by business process simulation concepts. The Palladio meta-model can easily be extended by a process model and an organization environment model, as it is already structured in several sub models on different layers such as hardware layer or software component layer. In contrast to ADONIS, Palladio provides an Eclipse-based open source tool support. A lot of existing simulation behavior can be reused or easily be adapted for the new process elements as actors and IT resource often behave similarly while processing jobs. A first sketch of an integrated simulation of business processes and IT systems as well as tool support is presented in [HHP12].

8. Conclusion

In this paper, we presented a simulation-based approach to predict the impact of a business process design on the performance of supporting IT systems and vice versa. We discussed how the mutual impact of business processes and IT systems is reflected by isolated simulations. We concluded that when using process simulation and IT simulation in isolation, workload distribution within the process is not represented correctly which can significantly reduce the performance prediction accuracy. Even so, our approach is helpful in many cases where rough estimations are sufficient. We argue for an integrated simulation to achieve a higher accuracy as there are several analogies in business process simulation and IT simulation. We are currently extending Palladio by business process simulation concepts in order to realize an integrated simulation of business processes and IT systems. In a case study, we are currently evaluating the feasibility and acceptance of our approach in practice. In the future, we want to evaluate the prediction accuracy of the integrated simulation by comparing it to the approach presented in this paper as well as to data measured in reality.

References

[Bet12] Betz, S.; Burger, E.; Eckert, A.; Oberweis, A.; Reussner, R.; Trunko, R.: An approach for integrated lifecycle management for business processes and business software. IGI Global, 2012.

[BKR09] Becker, S.; Koziolek, H.; Reussner, R.: The Palladio component model for model-driven performance prediction, JSS, vol. 82, pp. 3–22, 2009.

[Bol98] Bolch, G.; Greiner, S.; de Meer, H.; Trivedi, K.S.: Queueing networks and Markov chains: modeling and performance evaluation with computer science applications, Wiley-Interscience, 1998.

[GPK99] Giaglis, G.M.; Paul, R.J.; O Keefe, R.M.: Research note: Integrating business and network simulation models for it investment evaluation. Logistics Information Management, Vol. 12(1-2), pp. 108-117, 1999.

[HHP12] Heinrich, R., Henss, J., Paech, B.: Extending Palladio by Business Process Simulation Concepts, in Palladio Days 2012 Proceedings, ISSN 2190-4782, pp.19-27, 2012.

[HJK97] Herbst, J., Junginger, S., and Kühn, H.: Simulation in Financial Services with the Business Process Management System ADONIS, in Proceedings of the 9th European Simulation Symposium, pp. 491-495, 1997.

[Mi08] Mi, N.; Casale, G.; Cherkasova, L.; Smirni, E.: Burstiness in multitier applications: symptoms, causes, and new models, in Proceedings of the 9th International Conference on Middleware, Springer-Verlag, pp. 265–286, 2008.

[Oxf12] Oxford dictionaries online, Available: http://oxforddictionaries.com/

[Pai96] Painter, M.K.; Fernandes, R.; Padmanaban, N.; Mayer, R.J.: A Methodology for Integrating Business Process and Information Infrastructure Models. WSC, IEEE, pp. 1305-1312, 1996.

[Sav12] Jboss community, Savara, Available: http://www.jboss.org/savara.

[SH05] Serrano, A.; den Hengst, M.: Modelling the Integration of BP and IT Using Business Process Simulation. Enterprise Information Management, 18(6), pp. 740-759, 2005.

[TT07] Tan, Y.; Takakuwa, S.: Predicting the Impact on Business Performance of Enhanced Information System Using Business Process Simulation, WSC, pp. 2203-2211, 2007.

[WMC99] Workflow Management Coalition: Document No. WFMC-TC-1011, 1999.

Patchen von Modellen

Udo Kelter, Timo Kehrer, Dennis Koch
Praktische Informatik
Universität Siegen
{kelter,kehrer,dkoch}@informatik.uni-siegen.de

Abstract: Für die modellbasierte Softwareentwicklung werden spezialisierte Werkzeuge für ein professionelles Versions- und Variantenmanagement von Modellen benötigt. Insbesondere Anwendungsfälle wie das Patchen oder Mischen von Modellen stellen sehr hohe Anforderungen an die Konsistenz der synthetisierten Modelle. Während für das klassische 3-Wege-Mischen von Modellen bereits erste brauchbare Ansätze vorgeschlagen wurden, bestehen erhebliche Defizite bei Patch-Werkzeugen. Anhand realer Einsatzszenarien von Patch-Werkzeugen erörtern wir die wesentlichen Anforderungen und Schwierigkeiten und analysieren potentielle Fehlerquellen bei der Anwendung von Patches. Daraus leiten wir wesentliche Entwurfsentscheidungen zur Konstruktion von Patch-Werkzeugen ab und stellen unseren Ansatz zum konsistenzerhaltenden Patchen vor.

1 Einleitung und Motivation

Die modellbasierte Softwareentwicklung leidet nach wie vor an einer unzureichenden Unterstützung durch Versionsmanagement-Werkzeuge [EM12]. Während teilweise schon brauchbare Werkzeuge zum Anzeigen von Differenzen und Mischen von Modellen vorhanden sind, bestehen erhebliche Defizite bei Patch-Werkzeugen. Die verfügbaren Werkzeuge sind nicht annähernd so ausgereift und breit einsetzbar wie z.B. die UNIX-Standardwerkzeuge `patch` oder `diff`.

Dieses Papier behandelt das Patchen von Modellen. Patchen (*to patch* - ausbessern, flicken, korrigieren) bezeichnet ganz allgemein den Vorgang, ein Dokument durch Anwendung einer Änderungsvorschrift (*the patch* - der Flicken, die Korrektur) abzuändern. Ein Patch entsteht i.d.R. durch Berechnung der Differenz zwischen zwei Dokumenten; er besteht aus einer Sequenz von Editierschritten, die das erste Dokument in das zweite überführen. Abschnitt 2 definiert grundlegende Begriffe und grenzt das Patchen vom 3-Wege-Mischen ab.

Patch-Werkzeuge werden für mehrere Entwicklungsaufgaben benötigt, bei denen unterschiedliche Randbedingungen zu beachten sind (s. Abschnitt 3). Das Hauptproblem beim Patchen ist die Anwendung eines Patches auf ein anderes Modell als das, das beim Vergleich als Basismodell diente: hierdurch kann die Anwendung des Patches scheitern und das Modell so inkorrekt werden, daß es nicht mehr mit Modelleditoren verarbeitet werden kann. Abschnitt 4 analysiert diese Probleme und führt die in diesem Papier unterstellte Definition korrektheitserhaltender Editieroperationen ein.

Hauptbeitrag des Papiers ist ein Konzept zum korrektheitserhaltenden Patchen von Modellen. Es unterstellt einen Satz korrektheitserhaltender Editieroperationen auf Modellen und ein Modellvergleichsverfahren wie z.B. in [KKT11] vorgestellt, das Differenzen mit Hilfe dieser Editieroperationen darstellt. Für so gewonnene Patches wird ein Verfahren und zugehöriges Werkzeugdesign vorgestellt, wie ein Patch kontrolliert auf einem Zielmodell angewandt werden kann. Hierbei werden diverse Fehlerfälle behandelt und Möglichkeiten für manuelle Eingriffe angeboten. Details sowie eine beispielhafte Benutzung der prototypischen Implementierung des Patchwerkzeugs werden in Abschnitt 5 vorgestellt. Abschnitt 6 vergleicht das hier vorgestellte Design mit anderen Ansätzen und Abschnitt 7 faßt die wesentlichen Erkenntnisse zusammen.

2 Grundlegende Begriffe

Ein **Patch** ist eine halbgeordnete Menge von Editierschritten (die üblicherweise in einer mit der Halbordnung konsistenten linearen Ordnung notiert bzw. gespeichert wird).

Ein **Editierschritt** ist analog zu einem Statement in einem Programm ein Aufruf einer Editieroperation mit passenden Parametern, darunter immer wenigstens eine Referenz auf ein Dokumentelement. Die Menge der zulässigen Editieroperationen hängt vom Dokumenttyp ab und wird als dessen **Editierdatentyp** bezeichnet.

Die **Anwendung** eines Patches auf ein Dokument, das in diesem Zusammenhang auch als das **Zieldokument** (*target document*) bezeichnet wird, besteht darin, die in dem Patch enthaltenen Editierschritte auf dem Zieldokument auszuführen. Hierzu wird i.d.R. ein Interpreter benutzt, der Implementierungen der Editieroperationen beinhaltet. Bei der Anwendung eines Patches können diverse Fehler auftreten, die vom Einsatzszenario abhängen.

Erzeugt wird ein Patch normalerweise durch Berechnen einer Differenz zwischen zwei Dokumentversionen v0 und v1 und eine eventuelle Nachverarbeitung der initial gewonnenen Differenz[1]; v0 wird in diesem Fall auch als **Basisversion** bezeichnet, v1 als **geänderte Version**.

Bild 1 illustriert die Gewinnung und Anwendung eines Patches: zunächst werden die Dokumentversionen v0 und v1, die aus Repository 1 stammen, verglichen; die resultierende Differenz wird zu einem Patch verarbeitet. Der Patch soll nun auf das Zieldokument vt angewandt werden, das potentiell aus einem anderen Repository stammen kann. Hierzu wird vt in den Workspace 1 ausgecheckt, der Patch dort angewandt und das veränderte Dokument als Nachfolgeversion von vt wieder eingecheckt.

Das Patchen wird oft auch als “Mischen” bezeichnet, arbeitet aber deutlich anders als das 3-Wege-Mischen[2]. Das 3-Wege-Mischen ist in Bild 1 in Workspace 2 illustriert: gemischt werden die Version v1, die sich im Repository befindet, und die modifizierte Kopie von v0 im Workspace 2, hier als v2 bezeichnet; v1 und v2 haben beide v0 als gemeinsame Basisversion. Die beiden Differenzen diff(v0,v1) und diff(v0,v2) können Editierschritte

[1] Allerdings sind auch andere Methoden denkbar, z.B. eine manuelle Erstellung oder die Protokollierung von Änderungsoperationen in syntaxbasierten Editoren offener Entwicklungsumgebungen [SZN04, HK10]

[2] Auf das 2-Wege-Mischen gehen wir hier nicht ein.

beinhalten, die unverträglich sind, d.h. einer der Editierschritte muß “geopfert” oder manuell modifiziert werden. Paare derartiger Editierschritte, bei denen keine automatisierte Entscheidung möglich ist, werden als **Konflikt** bezeichnet.

Beim Patchen gibt es i.a. keine gemeinsame Basisversion, v0 und vt können völlig unkorreliert sein und in verschiedenen Repositories liegen. Der Begriff Konflikt ist daher hier nicht anwendbar; man könnte zwar die Differenz diff(v0,vt) bilden und hätte dann scheinbar die 3-Wege-Situation mit v0 als Basisversion, die Differenz diff(v0,vt) steht aber nicht zur Disposition, Editierschritte daraus können nicht geopfert werden. Ferner steht bei manchen Einsatzszenarien (s. Abschnitt 3) die Version v0 gar nicht für einen Vergleich zur Verfügung. Wenn überhaupt, können nur Editierschritte im Patch geopfert werden.

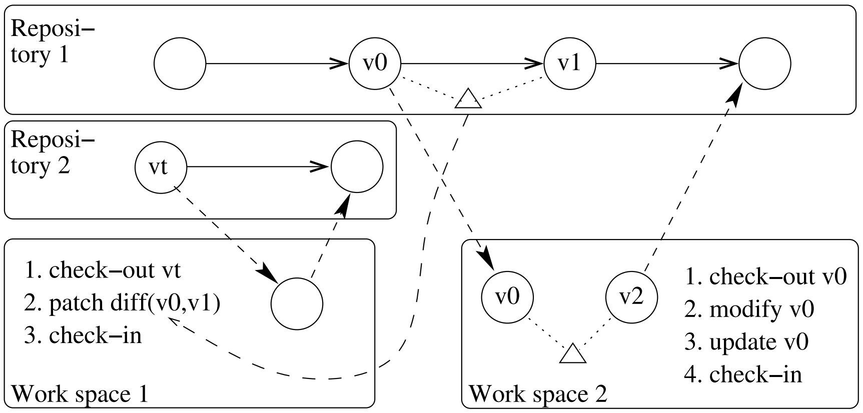


Abbildung 1: Patching vs. 3-Wege-Mischen

Wird ein auf diff(v0,v1) basierender Patch auf v0 angewandt, entsteht wieder v1, d.h. man kann mit Hilfe des Patches aus v0 die Version v1 rekonstruieren. Die Umkehrung gilt i.a. nicht, weil man aus einem Editierschritt nicht ohne weiteres den inversen Editierschritt ableiten kann. Daher werden Patches auch als **asymmetrische Differenz**, **directed delta** [Me02] oder **edit scripts** bezeichnet.

3 Einsatzszenarien von Patch-Werkzeugen

Das Patchen von Softwaredokumenten hat eine lange Tradition bei textuellen Dokumenten; die entsprechenden Einsatzszenarien ergeben sich analog für Modelle:

(1) Abgleich identischer Kopien eines Dokuments: Wenn z.B. ein Dokument auf mehreren Rechnern identisch vorhanden ist, müssen die Kopien im Rahmen eines Rollouts eines neuen Software-Releases auf einen neuen Stand gebracht werden. Man könnte theoretisch auch die kompletten Dokumente verschicken, Patches sind aber i.a. wesentlich kleiner; der Hauptvorteil ist daher die Reduktion des Transportvolumens. Ein äquivalentes

Einsatzszenario ist die Delta-Speicherung einer Revisionshistorie in einem Repository. Hauptmerkmal dieser Einsatzszenarien ist, daß der Patch immer nur auf eine Kopie seiner Basisversion angewandt wird und daher ohne Fehlerfälle abgearbeitet werden kann.

(2) "Cherrypicking changes" [CFP11]: Unterstellt wird hier typischerweise ein Versionsgraph wie in Bild 1 dargestellt. Es liegen zwei parallele Zweige vor, im unteren Zweig, in dem vt liegt, sollen selektiv die Änderungen, die im anderen Zweig zwischen v0 und v1 auftraten, übernommen werden. v0 und vt müssen nicht notwendig einen gemeinsamen Vorgänger haben, sondern können in verschiedenen Repositories liegen.

(3) Propagation von Änderungen in einer Menge von Varianten: In der Praxis werden vielfach Software-Systemfamilien als Mengen von autarken Varianten gehandhabt, namentlich in einer Anfangsphase, bevor Methoden des Product Line Engineering eingeführt werden. Eine Verbesserung in einer Variante, z.B. eine Fehlerkorrektur oder ein neues Feature, muß dann auf alle anderen Varianten propagiert werden. Technisch ähnelt dies stark dem Cherrypicking: die Verbesserung stellt sich als Patch dar, der durch Vergleich von Revisionen in der Ursprungsvariante gebildet wird und der auf *mehrere* Ziel-Dokumente angewendet werden soll.

4 Fehlerquellen bei der Anwendung von Patches

4.1 Identifikation von Operationsargumenten

Bei der Anwendung eines Patches müssen die Referenzen auf die Dokumentelemente aufgelöst werden, welche als Argumente von Editierschritten verwendet werden; jeder Referenz muß ein konkretes Element im Zieldokument zugeordnet werden.

Sofern ein Patch auf seine Basisversion oder eine Kopie hiervon angewandt wird (s. Abschnitt 3, Einsatzszenario 1), ist die Auflösung der Referenzen unproblematisch; als Referenzen können hier Zeilennummern, eindeutige Pfadnamen oder diverse andere Daten benutzt werden, mit denen die zu ändernden Dokumentelemente zuverlässig wiedergefunden werden.

Andernfalls (s. Abschnitt 3, Einsatzszenarien 2 und 3) steht man vor zwei gravierenden Problemen bei der Auflösung von Referenzen:

1. Ein referenziertes Dokumentelement ist ggf. nicht vorhanden, z.B. weil es gelöscht wurde. Der entsprechende Editierschritt ist somit prinzipiell **nicht ausführbar**.
2. Das referenzierte Dokumentelement befindet sich im Zieldokument an einer anderen "Position" als im Basisdokument; d.h. die im Patch angegebene Referenz, die letztlich ein Datenwert ist, der ein Modellelement identifizieren kann, muß als inkorrekt erkannt und wenn möglich korrigiert werden. Offensichtlich besteht hier die Gefahr **der Falschausführung von Editierschritten**, weil die Inkorrektheit der Referenz nicht er-

kannt wird oder der Korrekturversuch fehlschlägt und somit der Editierschritt an einem falschen Dokumentelement ausgeführt wird.

Die vorstehenden Fehlerquellen stellen schon bei textuellen Dokumenten ein Problem dar, bei Modellen werden sie durch die einzuhaltenden Korrektheits- bzw. Konsistenzkriterien wesentlich vergrößert.

4.2 Korrektheitsgrade von Modellen und korrektheitserhaltende Editieroperationen

Konzeptuell wird ein Modell als abstrakter Syntaxgraph (**ASG**) betrachtet, dessen Knoten, Kanten und Attribute durch ein Metamodell spezifiziert sind. Neben der elementaren Syntax spezifiziert das Metamodell oft zusätzliche, nicht-lokal wirksame Konsistenzbedingungen (z.B. mit Hilfe der OCL).

Die Metamodelle von Modellierungssprachen spezifizieren üblicherweise "weitgehend korrekte" Modelle, die zumindest so korrekt sind, daß man die übliche graphische Darstellung generieren kann. In der Praxis weichen viele Editoren von den Standards ab, sowohl durch zusätzliche Korrektheitskriterien wie durch nicht beachtete. Im Endeffekt entscheidend ist der minimale Korrektheitsgrad, den Modelle einhalten müssen, um überhaupt von Editoren weiterverarbeitbar zu sein. Patchwerkzeuge müssen sicherstellen, daß gepatchte Modelle diesen minimalen Korrektheitsgrad aufweisen und in dieser Hinsicht an die jeweiligen Modelleditoren in einer Entwicklungsumgebung angepaßt sein. Dieser minimale Korrektheitsgrad führt zu mehreren unangenehmen Konsequenzen:

1. *Nichttriviale minimale korrektheitserhaltende Editieroperationen*: Einzelne elementare (generische) Graphmodifikationen können aus Benutzersicht sinnlos sein und einen nicht mehr graphisch anzeigbaren, inkonsistenten Zustand erzeugen [KKT11]. Selbst wenn in den meisten Fällen elementare Graphmodifikationen erlaubt sind (z.B. das Setzen von Namen), treten Fälle auf, in denen ein korrektheitserhaltender Zustandsübergang nur durch mehrere elementare Graphmodifikationen bewirkt werden kann. *Minimale korrektheitserhaltende Editieroperationen* bestehen daher i.a. aus mehreren elementaren Graphmodifikationen, die analog zu einer Transaktion einen bisherigen konsistenten Zustand in einen anderen überführen und nur ganz oder gar nicht ausgeführt werden dürfen.

   Die zulässigen Editieroperationen müssen für jeden einzelnen Modellelementtyp individuell bestimmt werden. I.f. gehen wir davon aus, daß ein geeigneter Editierdatentyp bestimmt wurde.

2. *Zustandsabhängige Ausführbarkeit*: Eine korrektheitserhaltende Editieroperationen ist ggf. nur ausführbar, wenn bestimmte Bedingungen erfüllt sind[3]. Beispielsweise darf in einem Zustandsautomaten kein Startzustand erzeugt werden, wenn schon einer vorhanden ist, bei der Erzeugung einer Vererbungsbeziehung in einem Klassendiagramm darf

[3]Im Gegensatz dazu sind Editieroperationen auf Texten immer ausführbar, da es (normalerweise) keine Vorbedingungen oder Konsistenzkriterien gibt.

kein Zyklus entstehen usw. Die Ausführung einer Editieroperation kann also abhängig vom Modellzustand scheitern.

Durch scheiternde Editieroperationen entsteht (neben fehlenden Argumenten) eine neue Kategorie von Ursachen, warum einzelne Editierschritte eines Patches auf einem Zielmodell nicht ausführbar sein können.

5 Korrektheitserhaltendes Patchen

In Abschnitt 4 wurden mögliche Fehlerquellen analysiert, welche bei der Anwendung eines Patches auf ein von seiner Basisversion abweichendes Zielmodell auftreten können. Insbesondere bei der Propagation von Änderungen in einer Systemfamilie ist für jedes der Zielmodelle im Einzelfall zu prüfen, ob die Änderung hier sinnvoll und zulässig ist, speziell bei sicherheitskritischer eingebetteter Software. Daher wird man hier Wert auf Werkzeugfunktionen legen, mit denen man (a) die Auswirkungen eines Patches kontrollieren kann[4], (b) bei Bedarf den Patch sinnvoll modifizieren und in dieser Form archivieren kann und (c) die Anwendung des Patches steuern kann, z.B. durch manuelle Vorgaben, wie Referenzen auf Operationsargumente aufzulösen sind. Derartige Werkzeugfunktionen können auch beim Einsatzszenario Cherrypicking sinnvoll sein, werden aber von derzeit verfügbaren Werkzeugen nicht angeboten.

Das hier vorgestellte Konzept basiert auf vier separaten Werkzeugen bzw. integrierten Werkzeugfunktionen: einem Modellvergleichswerkzeug, das asymmetrische Differenzen auf Basis korrektheitserhaltender Editieroperationen erzeugt (s. Abschnitt 5.1), einem erweiterten Differenzanzeigewerkzeug zum Editieren von Patches (s. Abschnitt 5.3), der Matching-Funktion eines Modellvergleichwerkzeugs zur Auflösung von Referenzen auf Modellelemente im Zielmodell (s. Abschnitt 5.4) und einem interaktiven Werkzeug zur kontrollierten Anwendung von Patches (s. Abschnitt 5.5).

Unsere prototypische Implementierung basiert auf dem Eclipse Modeling Framework. Als Transformationstechnologie wird das auf Graphtransformationstechniken beruhende Modelltransformationssystem EMF Henshin [Ar10] verwendet.

5.1 Bilden von Patches

Erzeugt wird ein Patch durch Vergleich des Ursprungsmodells mit dem geänderten Modell. Gängige Verfahren [KRPP09] liefern im Kern jedoch *symmetrische* Differenzen [Me02], d.h. eine Menge von Korrespondenzen, die "gleiche" Elemente der beiden Modelle einander zuordnen. Indem eines der Modelle zum Basismodell erklärt wird, werden Modellelemente, die im anderen Modell nicht mehr bzw. neu vorhanden sind, als *gelöscht* bzw.

[4]Die Auswirkungen eines Patches kann man ggf. kontrollieren, indem man das Zieldokument in einem Workspace auscheckt, den Patch anwendet und das modifizierte Dokument mit dem Original vergleicht. Diese Notlösung scheitert aber, sofern der Patch nicht erfolgreich anwendbar ist.

erzeugt angesehen. Hierdurch entsteht eine *asymmetrische* Differenz vom Basismodell zum geänderten Modell, welche jedoch beliebig viele “low-level” Editierschritte enthalten kann, die potentiell zu inkonsistenten Zwischenzuständen führen.

Wir nutzen hier das in [KKT11] vorgestellte Verfahren, welches in einem Folgeschritt Gruppen von low-level Änderungen identifiziert, die korrektheitserhaltende Editieroperationen realisieren (s. Bild 2). Der in [KKT11] vorgestellte regelbasierte Ansatz garantiert ferner die Konsistenz der erkannten Editierschritte und der definierten Editieroperationen, da für die Operationserkennung benötigte Erkennungsregeln automatisch auf Basis von Editierregeln generiert werden. Es wird also kein Aufruf einer Editieroperation erkannt, welche nicht definiert ist.

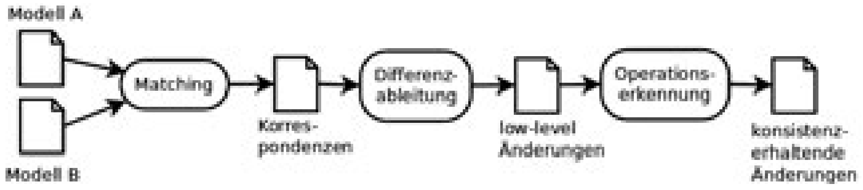


Abbildung 2: Struktur zustandsbasierter Differenzberechnungen

Für die Zwecke der Patchbestimmung wurde dieses Verfahren an mehreren Stellen erweitert, die für eine reine Differenzdarstellung nicht notwendig sind: (a) die Spezifikation der Editieroperationen wurde um Ein- und Ausgabeparameter ergänzt; (b) beim Bilden der Gruppen werden nun den formalen Parametern konkrete Werte, insb. Referenzen auf Modellelemente, zugeordnet; (c) Abhängigkeiten zwischen Editierschritten, die durch Ein- und Ausgabewerte entstehen, werden verwaltet.

5.2 Repräsentation von Patches

Unabhängig von dem zur Berechnung asymmetrischer Differenzen eingesetzten Verfahren ist von entscheidender Bedeutung, daß ein Patch bei Bedarf modifiziert und in dieser Form archiviert werden kann. Aus werkzeugtechnischer Sicht wird hierdurch ein Patch zu einem *editierbaren Dokument* und muss in geeigneter Form repräsentiert werden.

In der Literatur vorgeschlagene Datenmodelle zur Repräsentation von Modelldifferenzen entstammen dem Kontext des zustandsbasierten Vergleichs und basieren meist auf low-level Änderungen [CRP07, RV08], welche keinerlei Konsistenzkriterien beachten. Die Gruppierung von low-level Änderungen zu konsistenzerhaltenden Editierschritten ist zwar in wenigen Datenmodellen für Differenzen vorgesehen [KKT11, Ko10], Abhängigkeiten zwischen Editierschritten sowie aktuelle Aufrufparameter der entsprechenden Editieroperationen werden jedoch nicht explizit modelliert.

Ein Datenmodell zur Repräsentation asymmetrischer Differenzen EMF-basierter Modelle, welches den Anforderungen für das korrektheitserhaltende Patchen genügt, ist in Bild 3 dargestellt. Eine asymmetrische Differenz (*AsymmetricDifference*) zwischen einem Basismodell und einem geänderten Modell (jeweils repräsentiert durch ein EMF *ResourceSet*) besteht aus einer Menge von Editierschritten (*OperationInvocation*), welche durch eine Menge zyklenfreier Abhängigkeitsbeziehungen (*Dependency*) halbgeordnet wird.

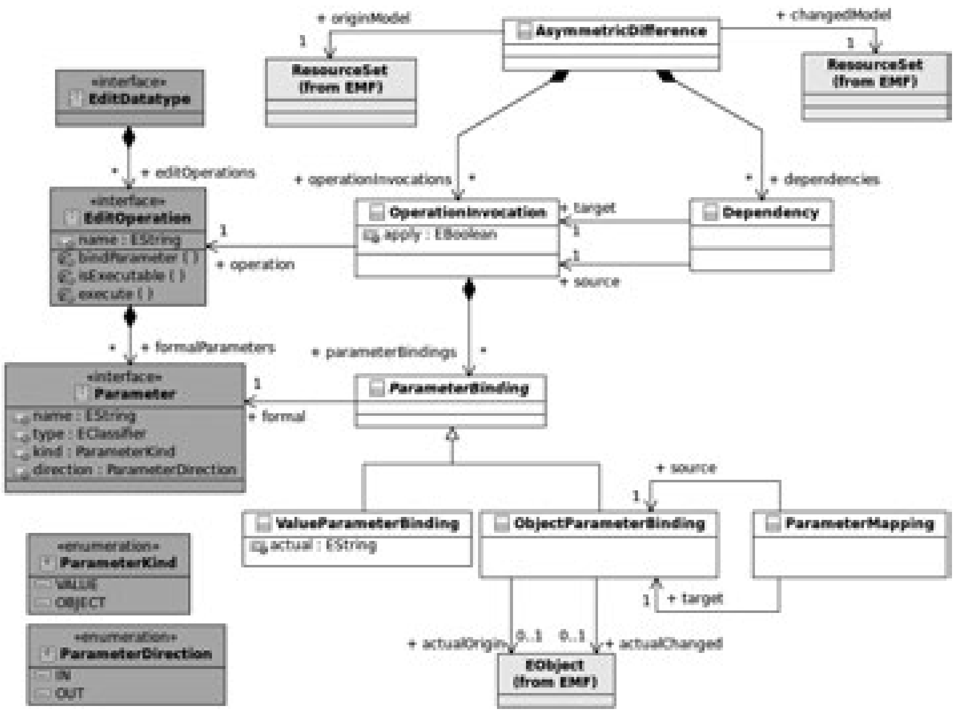


Abbildung 3: EMF-basierte Repräsentation von Patches

Einem Editierschritt wird die Signaturdeklaration der aufgerufenen Editieroperation (*EditOperation*) des zugrundeliegenden Editierdatentyps (*EditDatatype*) zugeordnet. An jeden formalen Parameter (*Parameter*) einer Editieroperation werden die aktuellen Parameterbelegungen gebunden (*ParameterBinding*): An Wertparameter (*kind = ParameterKind::VALUE*) wird die String-Repräsentation des entsprechenden Datenwerts gebunden (*ValueParameterBinding.actual*). An Objektparameter (*kind = Parameter-Kind::OBJECT*) werden Objekte der internen Repräsentation (*EObject*) des Basismodells (*ObjectParameterBinding.actualOrigin*) bzw. des geänderten Modells (*ObjectParameter-Binding.actualChanged*) gebunden.

In einem Editierschritt geänderte Objekte existieren in beiden Modellversionen, gelöschte Objekte existieren lediglich in der Basisversion und erzeugte Objekte existieren nur in der geänderten Version. In einem Editierschritt erzeugte Objekte sind Argumente formaler Ausgabeparameter (*direction = ParameterKind::OUT*). In einem Editierschritt verwendete Objekte sind Argumente formaler Eingabeparameter (*direction = ParameterKind::IN*). Ausgabeargumente, welche in einem späteren Editierschritt als Eingabeargumente verwendet werden, werden entsprechend aufeinander abgebildet (*ParameterMapping*).

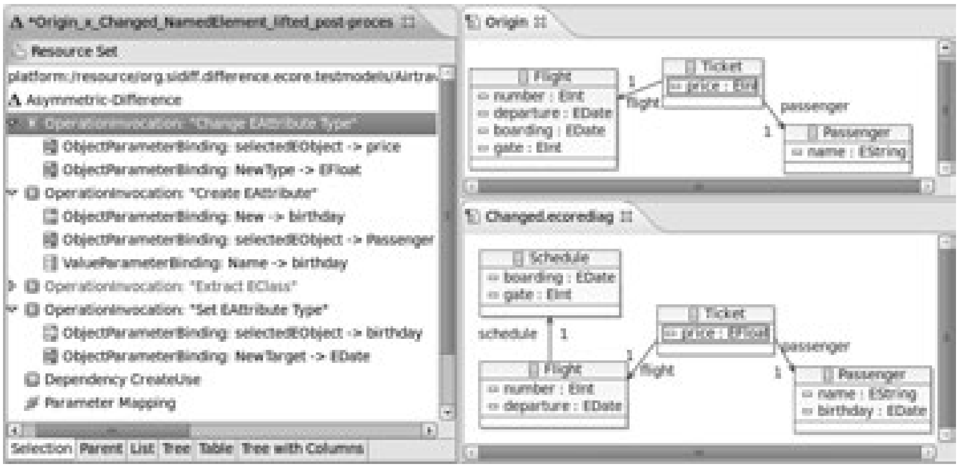
5.3 Editieren von Patches

Die Modifikation eines Patches vor seiner eigentlichen Anwendung besteht i.a. darin, den Patch auf eine Teilmenge der enthaltenen Editierschritte zu reduzieren. Die wesentliche Herausforderung besteht darin, daß der modifizierte Patch bei seiner Anwendung auf das Zielmodell korrektheitserhaltend ist.

Zum Editieren von Patches wird unterstellt, daß Ursprungsmodell und geändertes Modell zur Verfügung stehen. Ein Werkzeug zum Editieren von Patches kann somit auf Basis eines bestehenden Anzeigewerkzeugs für Differenzen realisiert werden, sofern dieses die Darstellung nicht-trivialer Editierschritte unterstützt, so z.B. das Darstellungskonzept der klickbaren Liste lokaler Unterschiede [KKOS12, KSW08]. Diese Anzeigeform ermöglicht die visuelle Darstellung des Kontexts eines Editierschritts, d.h. die aktuellen Parameterbelegungen. Dies ist insbesondere dann entscheidend, wenn “anonyme” Modellelemente, so z.B. arithmetische Operatoren in ASCET- oder Matlab/Simulink-Blockdiagrammen, als Operationsargumente verwendet werden.

Bild 4 zeigt die grafische Bedienschnittstelle unseres prototypischen Editors für Patches. Das Ursprungsmodell sowie das geänderte Modell werden jeweils in einem eigenen Editorfenster angezeigt. Das Beispielszenario zeigt zwei Revisionen eines in Ecore modellierten Entwurfsklassendiagramms für ein exemplarisches Flugbuchungssystem.

In einem Steuerfenster wird die klickbare Liste der in der Differenz enthaltenen Editierschritte angezeigt. Ein Editierschritt ist der angewendeten Editieroperation entsprechend benannt. Die lineare Anordnung der Editierschritte ist konsistent zur Halbordnung, welche aus den Abhängigkeiten der Editierschritte resultiert. Die von einem Editierschritt abhängigen Schritte können auf Wunsch visuell hervorgehoben werden. Die einzige Abhängigkeit im Beispiel aus Bild 4 besteht zwischen dem Erzeugen des Attributs *birthday* und dem anschließenden Setzen des Attributtyps auf den Datentyp *EDate*.



```
*Origin_x_Changed_NamedElement_lifted_post-proces
Resource Set
platform:/resource/org.sidiff.difference.ecore.testmodels/Airtra
Asymmetric-Difference
  OperationInvocation: "Change EAttribute Type"
      ObjectParameterBinding: selectedEObject -> price
      ObjectParameterBinding: NewType -> EFloat
  OperationInvocation: "Create EAttribute"
      ObjectParameterBinding: New -> birthday
      ObjectParameterBinding: selectedEObject -> Passenger
      ValueParameterBinding: Name -> birthday
  OperationInvocation: "Extract EClass"
  OperationInvocation: "Set EAttribute Type"
      ObjectParameterBinding: selectedEObject -> birthday
      ObjectParameterBinding: NewTarget -> EDate
  Dependency CreateUse
  Parameter Mapping
Selection | Parent | List | Tree | Table | Tree with Columns
```

Abbildung 4: Grafische Bedienschnittstelle zum Editieren von Patches

Ferner werden die aktuellen Aufrufparameter der einzelnen Editierschritte im Steuerfenster dargestellt. Objektparameter werden explizit als Eingabe- bzw. Ausgabeparameter gekennzeichnet, wobei aufeinander abgebildete Ein- und Ausgabeparameter farblich markiert werden, so im Falle des zuerst erzeugten und später verwendeten Attributs *birthday*. Wird ein Listeneintrag auf der linken Seite angeklickt, so werden die involvierten Modellelemente, d.h. die aktuellen Parameter eines Editierschritts, in den entsprechenden Editorfenstern fokussiert und farblich hervorgehoben.

Per Kontextmenü kann ein Benutzer einzelne Editierschritte aus dem Patch entfernen, davon abhängige Editierschritte werden ebenfalls entfernt, um den Patch konsistent zu halten. Die beiden in Bild 4 dargestellten Revisionen sollen einer Variante zur Ablaufplanung am Flughafen entstammen. Der Editierschritt zur Extraktion der Klasse *Schedule*, welcher lediglich variantenspezifische Informationen (das Abflug-Gate und die Boarding-Zeit) modifiziert, wurde daher aus dem Patch entfernt.

5.4 Auflösen von Referenzen auf Modellelemente

Zur Identifikation von Operationsargumenten im Zielmodell müssen die Referenzen auf Elemente des Ursprungsmodells aufgelöst werden. Hierzu kann der Matcher des Vergleichswerkzeugs (s. Bild 2) benutzt werden [5]: Die Basisversion v0 (s. Bild 1) wird mit der Zielversion vt verglichen, allerdings wird nur das Matching berechnet, die Differenzableitung und Operationserkennung entfallen. Das Matching ordnet letztlich den Elementen der Basisversion die entsprechenden Elemente im Zielmodell zu. Je nach Modelltyp und technischen Rahmenbedingungen können hier unterschiedliche Matching-Verfahren [KRPP09] eingesetzt werden.

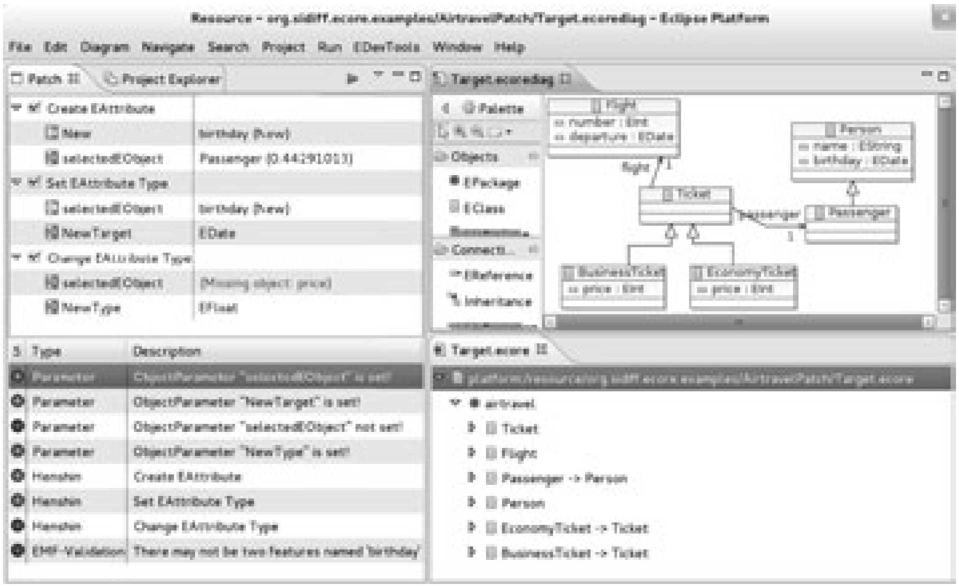
Bei allen Verfahren kann der Fall eintreten, daß einzelne Referenzen nicht aufgelöst werden können. Dies muss i.d.R. manuell bereinigt werden, indem die Referenz vom Entwickler aufgelöst wird oder der involvierte Editierschritt und alle von ihm direkt oder indirekt abhängigen Editierschritte im Patch gelöscht werden (s. Abschnitt 5.5).

Im Falle ähnlichkeitsbasierter Matcher besteht ferner die Gefahr der Falschauflösung von Referenzen. Eine Maßnahme zur Erkennung derartiger Fälle ist die Bewertung der Zuverlässigkeit von Korrespondenzen. Ein quantitatives Maß wird in [We11] vorgeschlagen, eine qualitative Bewertung auf Grundlage alternativer Matching-Kandidaten wird in [GK11] beschrieben. Beide Verfahren werden von dem von uns eingesetzten Vergleichswerkzeug SiDiff [KKPS12] unterstützt und integriert. So wird bspw. in dem in Abschnitt 5.5 dargestellten Anwendungsszenario die Zuverlässigkeit der Korrespondenz zwischen der Klasse *Passenger* des Ursprungsmodells und der Klasse *Passenger* des Zielmodells mit ca. 44% bewertet (s. Bild 5).

[5]Wir unterstellen hier, daß die Basisversion des Patches auf dem gleichen System wie die Zielversion verfügbar ist. Dies ist bei den Szenarien 2 und 3 typischerweise der Fall.

5.5 Kontrollierte Anwendung von Editierschritten

Nach der Auflösung der Referenzen im Zielmodell folgt die interaktive Überprüfung und kontrollierte Anwendung eines Patches. Abbildung 5 zeigt den initialen Zustand der interaktiven Anwendung des Patches unseres Beispielszenarios aus Abschnitt 5.3 auf eine vom Basismodell abweichende Variante des exemplarischen Flugbuchungssystems. Der Aufbau der Liste der anzuwendenden Editieroperationen (links oben) entspricht i.W. der Operationsliste des Editierwerkzeugs. Einzelne Editierschritte können bei Bedarf auch hier deselektiert werden, eine Konsistenzprüfung garantiert die automatische Deselektion abhängiger Editierschritte.



✓ Create EAttribute	
New	birthday (New)
selectedEObject	Passenger (0.44291013)
✓ Set EAttribute Type	
selectedEObject	birthday (New)
NewTarget	EDate
✓ Change EAttribute Type	
selectedEObject	[Missing object: price]
NewType	EFloat

S	Type	Description
●	Parameter	ObjectParameter "selectedEObject" is set!
●	Parameter	ObjectParameter "NewTarget" is set!
●	Parameter	ObjectParameter "selectedEObject" not set!
●	Parameter	ObjectParameter "NewType" is set!
●	Henshin	Create EAttribute
●	Henshin	Set EAttribute Type
●	Henshin	Change EAttribute Type
●	EMF-Validation	There may not be two features named 'birthday'

```
platform:/resource/org.sidiff.ecore.examples/AirtravelPatch/Target.ecore
  airtravel
    Ticket
    Flight
    Passenger -> Person
    Person
    EconomyTicket -> Ticket
    BusinessTicket -> Ticket
```

Abbildung 5: Interaktive Anwendung von Patches

Im Gegensatz zum Editierwerkzeug können Operationsargumente während der Anwendung des Patches explizit geändert (im Falle falsch aufgelöster Referenzen) oder gesetzt (im Falle nicht aufgelöster Referenzen) werden. Fehlende Operationsargumente werden hervorgehoben, so z.B. der Eingabeparameter für das Ändern des Typs des Attributs *price* unseres Ursprungsmodells[6]. Fehlende Argumente sind vom Benutzer zu selektieren und können ggf. im Zielmodell auch zunächst erstellt werden. Das Zielmodell unseres Anwendungsszenarios kann bspw. dahingehend refakturiert werden, daß die Attribute *BusinessTicket.price* und *EconomyTicket.price* durch ein Attribut *Ticket.price* der gemeinsamen Oberklasse ersetzt werden, welches anschließend als Eingabeparameter ausgewählt wird.

Die zustandsabhängige Ausführbarkeit einer Editieroperation wird überprüft, indem die Editieroperation versuchsweise auf einer internen Kopie des Zielmodells ausgeführt wird,

[6]Aufgrund von Uneindeutigkeit wurde im Beispielszenario keine Korrespondenz für dieses Attribut erzeugt

anschließend geprüft wird, ob Korrektheitsbedingungen verletzt wurden, und im Fehlerfall alle Änderungen zurückgesetzt werden. Zur Korrektheitsprüfung wird das EMF Validation Framework aufgerufen, eine entsprechende Implementierung der Überprüfung von Invarianten und Constraints wird vorausgesetzt. Validierungsfehler werden im Fehlerprotokoll (links unten) gemeldet. Im Beispielszenario schlägt die Erzeugung des Attributs *birthday* in der Klasse Passenger fehl, da ein gleichnamiges Attribut bereits durch die Oberklasse Person vererbt wird. Der entsprechende Editierschritt sowie der davon abhängige zum Setzen des Attributtyps kann vom Benutzer deselektiert werden, um die Fehlersituation aufzulösen.

Nach der Auflösung aller Fehlersituationen kann der modifizierte Patch auf das Zielmodell angewendet werden. Technisch ist hierfür jede aufzurufende Editieroperation, welche in der Repräsentation des Patches nur durch ihre Signatur im Sinne abstrakter Datentypen gegeben ist (vgl. Abbildung 3), durch eine Implementierung zu ersetzen. Konkrete Transformationstechnologien und deren ggf. vorhandene Einschränkungen werden somit gekapselt. In unserer prototypischen Implementierung verwenden wir hier die in EMF Henshin spezifizierten Editierregeln, welche bereits zur Operationserkennung (s. Abschnitt 5.1) benötigt werden. Die Ausführung einer Editierregel wird an den Henshin Regel-Interpreter delegiert, welcher die aktuellen Aufrufparameter einer Operation entgegen nimmt.

6 Vergleich mit anderen Arbeiten

Das Patchen von Modellen ist bisher erst in wenigen Projekten behandelt worden. Praktisch nutzbare Patchwerkzeuge sind zur Zeit kaum auffindbar.

Der in [CRP10] vorgestellte Ansatz fasst die Anwendung eines Patches als Graphtransformation auf und nutzt hierzu ein Graphtransformationssystem (ATL). Die bei der Anwendung eines Patches zu benutzenden Transformationsregeln werden aus einer Differenz zwischen zwei Modellen abgeleitet. Es wird ein sehr einfaches Modell von Differenzen unterstellt, das keine korrektheitserhaltenden Editieroperationen unterstützt. Das Problem, wie Differenzen als Basis von Patches gewonnen werden und wie Referenzen korrekt aufgelöst werden, wird nicht behandelt, hierzu werden nur Möglichkeiten aufgelistet. Es bleibt völlig unklar, wie der Fehlerfall, daß ein Patch nicht vollständig ausführbar ist, sicher erkannt und wie einem Entwickler eine brauchbare Darstellung der Fehlerursachen geliefert werden kann; dies würde einen tiefen Eingriff in den Regelinterpreter oder entsprechende Debugging-Interfaces erfordern. Das Ergebnismodell kann nach der Patchanwendung beliebig inkonsistent sein. Insgesamt ist der Ansatz für die in Abschnitt 3 beschriebenen Einsatzszenarien 2 und 3 unbrauchbar.

Ein deutlich praxisorientierteres Konzept für das Patchen von Modellen stellt Könemann [Ko10] vor. Vorgeschlagen wird ein detaillierter Prozeß, wie Roh-Differenzen zu einem Patch umgestaltet werden - hierzu werden mehrere naheliegende Heuristiken kombiniert - und wie der Patch später flexibel auf ein Zielmodell angewandt wird. Dieser Prozeß setzt an vielen Stellen auf interaktive Eingriffe von Entwicklern, u.a. um die Patch-Inhalte semantisch anzureichern und die Auflösung von Referenzen zu kontrollieren bzw.

zu korrigieren. Dieser Prozeß ist sehr flexibel, andererseits mit sehr hohem Arbeitsaufwand verbunden, da jede einzelne Gruppe von elementaren Änderung separat manuell behandelt werden muß. Das in unserem Ansatz unterstellte automatisierte Liften von Roh-Differenzen deckt nur einen Teil der Fälle ab, diese aber präziser und weitaus weniger arbeitsaufwendig. Analoges gilt für die Anwendung einer Differenz: Diese kann beim Könemannschen Ansatz auf eine individuelle Reimplementierung der Intention von Änderungen hinauslaufen. Im Vergleich dazu ist der Prozeß der Anpassung an das Zielmodell bei unserem Ansatz stärker automatisiert und bietet mehr Sicherheit gegen Korrektheitsverletzungen, u.a. durch Ausnutzung zusätzlicher Informationen aus der Matching-Engine, um Operationsargumente für die anzuwendenden Editieroperationen zuverlässig zu bestimmen. Beide Ansätze können im Prinzip kombiniert werden, wobei unser Ansatz die mechanisch erkennbaren komplexen Editieroperationen sicherer und effizienter bearbeiten kann, während darüber hinausgehende komplexere Editiervorgänge mit dem Könemannschen Verfahren behandelt werden könnten.

EMF Compare [EC12] unterstützt lediglich den Anwendungsfall des 3-Wege-Mischens. In einigen Fällen läßt sich das Patchen zwar auf das 3-Wege Mischen zurückführen, grundlegende Einschränkungen, so z.B. die Notwendigkeit der Existenz einer gemeinsamen Basisversion, wurden bereits in Abschnitt 2 diskutiert. Ferner besteht hierbei keine Möglichkeit, erzeugte Patches zu editieren und so auf eine konsistenzerhaltende Teilmenge von Editierschritten zu reduzieren.

Der im Rahmen des EMF Compare-Projekts entwickelte EPatch-Ansatz [EP12] benutzt Xtext zur externen Darstellung von Patches. Die nicht-interaktive Anwendung eines Patches ist hier zwar im Gegensatz zum 3-Wege-Mischwerkzeug von EMF Compare nicht mehr an die Existenz einer gemeinsamen Basisversion gebunden, Objekte im Zielmodell werden jedoch über statische, i.d.R. auf persistenten XMI-Identifizierern basierenden Pfadnamen lokalisiert. Patches sind somit lediglich auf exakte Kopien des Ursprungsmodells anwendbar.

7 Zusammenfassung

Dieses Papier präsentiert einen neuen Ansatz zum Patchen von Modellen: ein Hauptmerkmal dieses Ansatzes sind Maßnahmen, die sicherstellen, daß die gepatchten Modelle korrekt und durch andere Werkzeuge weiterverarbeitbar sind.

Technisch unterscheidet sich unser Ansatz von früheren dadurch, daß systematisch “geliftete” Differenzen benutzt werden, die nicht mehr aus elementaren ASG-Modifikationen, sondern aus korrektheitserhaltenden Editieroperationen bestehen. Besonderer Wert wurde auf die Benutzerunterstützung bei der Anwendung und ggf. interaktiven Anpassung eines Patches auf ein Zielmodell gelegt: es werden detaillierte Beschreibungen von Fehlerursachen geliefert, ferner werden Patches als editierbare Dokumente unterstützt.

Literatur

[Ar10] Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. Intl. Conf. Model Driven Engineering Languages and Systems 2010, Oslo; LNCS 6394, Springer; 2010

[CRP07] Cicchetti, A.; Ruscio, D.; Pierantonio, A.: A Metamodel independent approach to difference representation; Journal of Object Technology 6(9); 2007

[CRP10] Cicchetti, A.; Ruscio, D.; Pierantonio, A.: Model Patches in Model-Driven Engineering; p.190-204 in: Models in Software Engineering - Workshops and Symposia at MODELS 2009; LNCS 6002, Springer; 2010

[CFP11] Collins-Sussman, B.; Fitzpatrick, B.W.; Pilato, C.M.: Version Control with Subversion For Subversion 1.7; http://svnbook.red-bean.com/en/1.7/svn-book.pdf; 2011

[EM12] Emanuelsson, P.: There is a strong need for diff/merge tools on models; Position paper, CVSM 2012, http://pi.informatik.uni-siegen.de/CVSM2012; 2012

[EP12] EPatch; http://wiki.eclipse.org/EMF_Compare/Epatch; 2012

[EC12] EMF Compare; http://www.eclipse.org/emf/compare/; 2012

[GK11] Gorek, G.; Kelter, U.: Abgleich von Teilmodellen in den frühen Entwicklungsphasen; p.123-134 in: Software Engineering 2011; LNI 183, GI; 2011

[HK10] Herrmannsdörfer, M.; Kögel, M.: Towards a Generic Operation Recorder for Model Evolution; p.76-81 in: Proc. 1st Intl. Workshop on Model Comparison in Practice, Malaga; ACM; 2010

[KKOS12] Kehrer, T.; Kelter, U.; Ohrndorf, M.; Sollbach, T.: Understanding Model Evolution through Semantically Lifting Model Differences with SiLift; p.638-641 in: Proc. 28th Intl. Conf. on Software Maintenance (ICSM 2012); IEEE CS; 2012

[KKT11] Kehrer, T.; Kelter, U.; Taentzer, G.: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning; p.163-172 in: Proc. 26th Intl. Conf. on Automated Software Engineering (ASE 2011); ACM; 2011

[KKPS12] Kehrer, T.; Kelter, U.; Pietsch, P., Schmidt, M.: Adaptability of Model Comparison Tools; in: Proc. 27th Intl. Conf. on Automated Software Engineering; 2012

[KSW08] Kelter, U.; Schmidt, M.; Wenzel, S.: Architekturen von Differenzwerkzeugen für Modelle; p.155-168 in: Software Engineering 2008; LNI 121, GI; 2008

[Ko10] Könemann, P.: Capturing the Intention of Model Changes; p.108-122 in: Proc. Intl. Conf. Model Driven Engineering Languages and Systems 2010, Oslo, Part I; LNCS 6394, Springer; 2010

[KRPP09] Kolovos, D.S.; Ruscio, D.D.; Pierantonio, A.; Paige, R.F.: Different Models for Model Matching: An Analysis Of Approaches To Support Model Differencing; p.1-6 in: Proc. 2009 ICSE Workshop on Comparison and Versioning of Software Models; IEEE; 2009

[Me02] Mens, T.: A state-of-the-art survey on software merging; IEEE Trans. Softw. Eng.; 28(5), p.449-462; 2002

[RV08] Rivera, J.E.; Vallecillo, A.: Representing and Operating with Model Differences; p.141-160 in: Proc. TOOLS EUROPE 2008; LNBIP 11, Springer; 2008

[SZN04] Schneider, Ch.; Zündorf, A.; Niere, J.: CoObRA - a small step for development tools to collaborative environments; ICSE Workshop on Directions in Software Engineering Environments; 2004

[We11] Wenzel, S.: Unique Identification of Elements in Evolving Models: Towards Fine-Grained Traceability in Model-Driven Engineering; Dissertation, Universität Siegen; URN: urn:nbn:de:hbz:467-5243; 2010

Improved Prediction of Non-functional Properties in Software Product Lines with Domain Context

Max Lillack[a], Johannes Müller[b], Ulrich W. Eisenecker[a]

[a] {lillack, eisenecker}@wifa.uni-leipzig.de, [b] jmue933@aucklanduni.ac.nz

Abstract: Software Product Lines (SPLs) enable software reuse by systematically managing commonalities and variability. Usually, commonalities and variability are expressed by features. Functional requirements of a software product are met by selecting appropriate features. However, selecting features also influences non-functional properties. To satisfy non-functional requirements of a software product, as well, the effect of a feature selection on non-functional properties has to be known. Often an SPL allows a vast number of valid products, which renders a test of non-functional properties on the basis of all valid products impractical. Recent research offers a solution to this problem: the effect of features on non-functional properties of software products is predicted by measuring in advance. A sample of feature configurations is generated, executed with a predefined benchmark, and then non-functional properties are measured. Based on these data a model is created that allows to predict non-functional properties of a software product before actually building it. However, in some domains contextual influences, such as input data, can heavily affect non-functional properties. We argue that the measurement of the effect of features on non-functional properties can be drastically improved by considering contextual influences of a domain. We study this assumption on input data as an example for a contextual influence and using an artificial but intuitive case study from the domain of compression algorithms. Our study shows that prediction accuracy of non-functional properties can be significantly improved.

1 Introduction

Software Product Line Engineering (SPLE) allows to leverage the reusability of assets and to efficiently develop new products of a common domain on the basis of an existing reuse infrastructure. It comprises at least two core processes, domain engineering (development for reuse) and application engineering (development with reuse). The set of products realized by these processes is a Software Product Line (SPL) [CN07].

Commonalities and variability of a domain are usually documented with variability models of which feature models are the most common [CGR$^+$12]. In feature models commonalities and variability of a domain are described in terms of features. A feature is any aspect of a software product that is relevant to at least one stakeholder [CE00]. Usually, the main focus of feature models is to describe the functional properties of an SPL [CHS08].

A valid combination of features describing a software product is called a configuration. During the configuration process a user of a feature model (e.g. a developer) can select certain features according to her requirements. With the selection of a feature the remaining variability is reduced. The resulting configuration describes the functional properties of a valid product of the SPL.

Non-functional properties such as performance, resource usage and scalability can be just as important to users as functional properties. They are an important part of user requirements and have to be taken into account as well. However, from feature selection alone it is not apparent which non-functional properties a resulting software product will have. Thus, resulting software products have to be measured to assess their non-functional properties.

In SPLE it is not sufficient to assess the non-functional properties of a single product, rather, all products of an SPL have to be considered. In case of a high number of possible products it is practically impossible to measure every product on its own with methods from traditional software engineering. A solution to this problem, suggested by recent research (cf. Section 2), is to measure a sample of possible configurations of an SPL with a benchmark to predict the impact of features on non-functional properties of products not built before. Since presence or absence of feature combinations can affect the impact of features on non-functional properties, the measurements have to cover such *feature interactions* as well. The result of this process is a *performance model* which maps the set of configurations to non-functional properties of the resulting products [TP12]. For each non-functional property, the performance model contains the impact of each feature, their interaction with other features and a function to aggregate these values.

Products of an SPL are often executed in different runtime environments and exposed to different types of input data, which could influence their behavior. We call these influences *contextual influences*. Our hypothesis is that contextual influences can have an impact on the effect of features on non-functional properties of the resulting products.

In this case, including contextual influences into the measurement of non-functional properties can increase the precision of their prediction. In the following, we report on the results of a rather artificial but intuitive case study from the domain of compression algorithms with input data as one example of contextual influences.

The objective of this study is, first, to demonstrate how to incorporate contextual influences into the measurement of non-functional properties (Section 4) and, second, to test our hypothesis and to see whether considering contextual influences can be useful at all (Section 5). For our study we used synthetic input data to measure their effects isolated from other effects. This should allow to observe clearly whether contextual influences can have an effect and whether more extensive studies with real-world SPLs are worthwhile. A conclusion in Section 6 wraps up the paper and provides an outlook to future work.

2 Related Work

Within the last couple of years a reasonable amount of work on modeling, estimating and predicting non-functional properties for SPLs has been published.

Some work recognizes that contextual influences can have an effect on non-functional properties [KR10, HBR$^+$10]. However, their focus is on distinct properties of contextual influences, whereas we argue that the variability of contextual influences in a specific domain has to be considered to improve the precision of estimated performance models.

Besides work which discusses contextual influences on non-functional properties of software products, there is also work concerned with approaches to capture and model information about non-functional properties. A process to model non-functional properties is presented by Tawhid et al. [TP12]. They consider UML profiles for modeling non-functional properties. On that basis predictions about non-functional properties are possible. However, their approach to gather information about non-functional properties is neither automatic nor does it consider contextual influences.

Sincero et al. discuss that information about non-functional properties can enrich the configuration process of SPLs [SSPS10]. They present an approach to detect general influences of features on non-functional properties, but do not build a performance model on top of the gathered data. The approach is demonstrated with a subset of the features of the Linux kernel without considering any contextual influences on non-functional properties.

Recent work of Siegmund et al. [SKK$^+$12] and Guo et al. [GCA$^+$12] specifically focuses on building performance models for SPLs. Siegmund et al. present a process from gathering data to estimating a performance model and discuss techniques for the involved steps. The discussed technique for estimating the performance model is based on linear regression. Guo et al. follow a slightly different approach by applying classification and regression trees (CARTs). Both do not consider the effect of contextual influences on non-functional properties.

Most of the related work creates performance models based on functional features. They may therefore only be valid within the same context in which they were created. Since Siegmund et al. present the most comprehensive approach, we are going to extend their approach to consider contextual influences in estimating performance models for SPLs.

3 Background: Predicting Non-Functional Properties in SPLs

In their approach [SKK$^+$12, SRK$^+$12] Siegmund et al. select a subset of possible configurations of an SPL, generate corresponding products and measure their non-functional properties. They use the resulting sample to calculate the impact of each feature on non-functional properties. As a basic requirement the selection of configurations has to allow the isolation of each feature's effect. With the isolated effects of the features it is possible to predict non-functional properties for an arbitrary configuration by aggregating the effects of the containing features.

However, sometimes features do not only have an isolated effect on non-functional properties but interact with other features. In such cases the effect of one feature changes if an interacting feature is contained in a configuration, too. A performance model has to take feature interactions into account in order to realize sufficiently high prediction accuracy.

Feature interactions can be of different orders, and different techniques have to be applied to detect them.

A pairwise interaction (first order interaction) occurs if the presence of two features generates a different effect than the combined individual effects of the two features. For each feature there is a high number of other features with which they could possibly interact. To reduce the number of measurements, Siegmund et al. suggest the following heuristic: calculate a configuration with as many features as possible, measure the effect on a non-functional property and compare it with the expected effect on the non-functional property from the aggregated isolated effects of the contained features. If the measured effect does not deviate from the expected effect, the absence of a feature interaction is assumed. Otherwise, pairs of interacting features have to be identified.

This is done with the pairwise covering heuristic [POS+11]. The heuristic includes as many configurations as required to cover all combinations of two features. A measurement of a feature interaction between features A and B can be skipped if A implies B [SKK+12]. The measurement of A alone will always include the possible interaction effect. Such constraints reduce the set of measurements.

Second order interactions are present if the presence of three features generate another effect than the presence of only one or two of them. A high number of measurements are required to cover all possible second order interactions. Siegmund et al. consider second order interactions only if pairwise interactions exist between each of the three possible features. Based on the assumption that few features interact with many others, they also test features, which already have taken part in many pairwise interactions for second order interactions. Although higher order interactions are possible, they are not considered to limit the number of measurements and hence make the approach practicable [SRK+12].

The approach is extensively evaluated with case studies using SPLs from different domains (e.g. database, compiler and video encoding) and different non-functional properties (e.g. footprint and performance) [SKK+12, SRK+12]. The performance model is based on standard benchmarks applicable to the domains of the analyzed SPLs. The results show a high prediction accuracy in the evaluation with data of the utilized standard benchmarks.

4 Input Data in Predicting Non-functional Properties

Our hypothesis is: considering input data in the measurement can improve the prediction of non-functional properties. To study this, we will first have to clarify how to consider input data in the measurement of non-functional properties. Three questions will have to be answered. First, how to capture variability of input data of a domain. Second, how to link the variability of input data to the variability of an SPL. Third, how to measure the effect of input data on non-functional properties.

A common tool to capture the variability of an SPL is feature modeling. Since feature models are essentially designed to capture the variability of any relevant concept [CE00] they are useful to model the variability of possible input data of a domain as well.

Feature models consist of features which are hierarchically modeled in feature diagrams.

An example of a feature diagram can be found in our case study in Figure 3. The root of the hierarchy is one concept node. It represents all possible products of an SPL. It is refined with subsequent nodes, the sub-features. These sub-features describe common and variable parts of the products of an SPL. Features can either be solitary or grouped. Mandatory solitary features are present in every product containing the parent feature.

Since features of input data describe a different concern than features of an SPL, it is natural to model both concerns separately. This simplifies the modeling of the input data features and prevents the SPL feature model from being bloated. Both models can easily be related, for example, by feature model references and transparently processed afterwards.

The variability of SPLs is documented in feature models by a domain analyst during the analysis phase of domain engineering. In the same way the domain analyst can capture the variability of input data.

As with functional features, identifying relevant data features and their appropriate abstraction is a problem of the domain engineering phase. For example, domain analysis can show the size of the input data as an important property. As it is not feasible to model every possible size as a single feature, clustering can be applied to get a feature *Size* with sub-features such as *10kB* and *20MB* in an *or-group* as representatives for a class of input sizes.

During the configuration process features are selected, based on user requirements. It is not necessary to distinguish between functional features and data features because interactions between them are possible. E.g. selecting a data feature could require selecting a functional feature.

By capturing the variability of input data in a feature model, the adaptation of the approach of Siegmund et al. is straightforward. The variability caused by contextual influences can be treated as the variability of the SPL. The performance model has to be extended so that it does not only contain the estimates of the effects of the functional features but also estimates for the effects of the input data features. However, to actually measure the effect of input data on non-functional properties the measurement environment has to be extended so that it generates test data according to the selected features of the input data feature model.

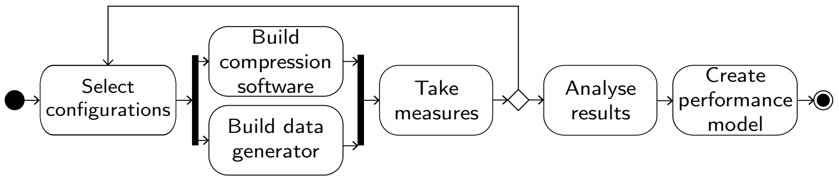


Figure 1: Activity diagram of measurement process

With that extension our measurement environment works according to the process depicted in Figure 1. The core of the measurement environment is a generator for products of the SPL, which are measured afterwards.

In contrast to the approach of Siegmund et al. [SKK$^+$12] we do not consider fixed input

data but generate different types of test data for different measurements. The products are executed with the test data and relevant non-functional properties are measured.
After all required measurements are collected they are analyzed and the performance model is created. A concrete implementation of this process is briefly described in Section 5.

For large SPLs it is infeasible to test every possible configuration because every test run needs time and other limited resources. Different configuration sets are built to isolate the effect of a single feature or a feature interaction from a complete configuration.

Based on the data produced by the measurement process, a performance model to predict non-functional properties of any configuration is constructed. We use a linear regression model as performance model. The model includes the effect of single features as well as interactions among any kind of features up to second order interactions. Features and their interactions are coded as binary variables (i.e. *dummy variables*) with the value 1 (present) or 0 (absent).

Statistical tools such as R[1] can be used to create the regression model with dummy coding. The regression model includes all features and valid feature interactions which can be identified by automatic analysis of feature models [BSRC10].

Siegmund et al. introduced a heuristic, where three-wise interactions are considered for three features which interact pairwise [SKK$^+$12]. This heuristic requires the reliable identification of pairwise interactions and assumes a low number of second order interactions. Depending on the structure of a feature model, a complete pairwise interaction detection could require a large amount of measurements.

We use a heuristic which increases the number of measurements of *interesting* features, i.e. features whose estimated direct effect or pairwise interaction effects are rather extreme. Based on the measurements of the pairwise set, we select results which exceed two predefined quantiles (e.g. 15% and 85% quantiles). Additional measurements are selected by replacing a sub-feature of a parent feature (e.g. in an or-group) with the other features of the same group holding the other features of the configuration constant. This heuristic is specific for each non-functional property.

With only little or no domain knowledge the heuristic can be adjusted to the requirements of the measurement environment. By adjusting the threshold set by the quantiles the number of measurements can be regulated. A higher number of configurations increases the time for measurements but improves the prediction quality. A second parameter is the selection of features to vary, the more features are fixed the less configurations need to be measured.

5 Study

In our study we test the hypothesis: considering input data can increase the precision of a performance model. Furthermore, we demonstrate our approach. In order to see the effect

[1]http://www.r-project.org

of input data on non-functional properties isolated from other effects we have designed an artificial SPL with lossless compression algorithms. The artificial nature of the study does limit its expressiveness but increases the clarity of the results. So, even if the study does not proof our hypothesis, its results provide a first indication whether our hypothesis should be further investigated with real-world SPLs.

We designed *Compressor SPL* (Figure 2) based on an already implemented compilation of compression tools[2] and an input data feature model (Figure 3). The Compressor SPL is similar to the *ZipMe* SPL [AKL12] but realizes more variants. The input data feature model comprises variability of input data, which is relevant for non-functional properties.

We considered three exemplary non-functional properties of lossless compression algorithms in the study: *compression time*, *memory usage* and *compression ratio*. Compression time is the time to finish one compression task with pre-specified input data. Memory usage is the amount of memory required during the compression. Compression ratio is the ratio between the size of the compressed data and the size of the input data.

We selected those three properties since they can be relevant in a decision for a specific compression algorithm. Though a low compression ratio is clearly desirable, a user can sacrifice ratio for improved speed (e.g. in time-critical applications) or lower memory consumption (e.g. in embedded systems). However, the selection of relevant properties solely relies on the addressed domain and has to be done in the domain analysis phase.

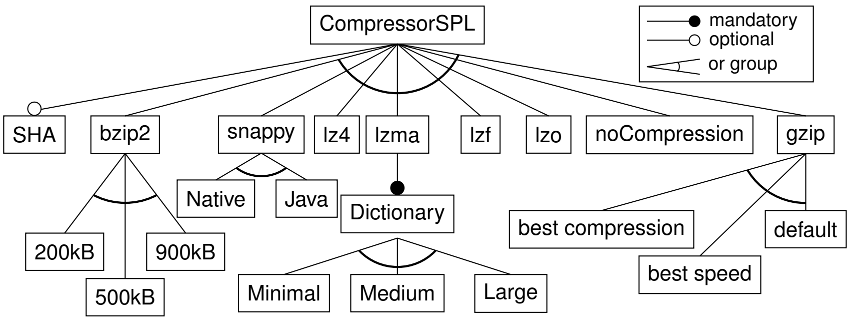


Figure 2: Feature Model of Compressor SPL

Figure 2 depicts the feature model of the Compressor SPL. It is basically a selection of different compression algorithms, implementations and configuration options. For example, the algorithm *bzip2* can be configured with different block sizes of which *200kB*, *500kB* and *900kB* are used as representative features. The block size is a popular user-configurable parameter which can have a great influence on performance [Sal07].

As another example the *snappy*[3] algorithm can have the feature *Native* or the feature *Java* to compare different implementations of the same algorithm.

Additionally there is an optional feature *SHA* to calculate a checksum of the input data using the SHA-256 hash function shared by the algorithms.

[2]https://github.com/ning/jvm-compressor-benchmark

[3]http://code.google.com/p/snappy

Figure 3 depicts the feature model of the input data of the domain. The model contains 14 features. It comprises two main features *Size* and *Content*. *Size* has six sub-features rep-

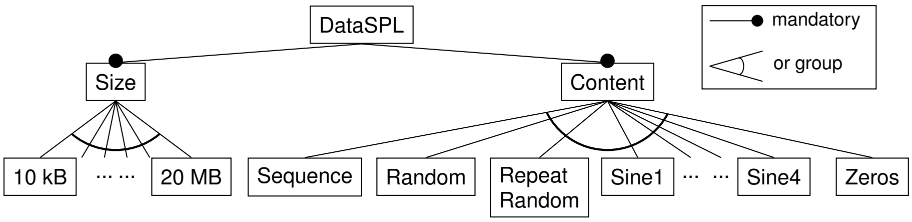


Figure 3: Feature Model of Data SPL

resenting different input data sizes from 10kB up to 20MB. The feature *Content* describes what kind of data is generated. It has several sub-features which represent prototypical structures of input data, which are differently difficult to compress. A *Sequence* is input data that contain sequentially changed bytes. The first thousand bytes are zeros, the next thousand bytes are ones and so on. *Random* is a sequence of random bytes. As a variant of *Random RepeatRandom* repeats small chunks of random bytes. *Sine1* to *Sine4* create non-trivial sequences based on different sine waves. *Zeros* contains only zeros.

For a transparent handling of the *Compressor SPL* feature model and the *Data SPL* feature model, we joined both in an artificial feature model, *Measurement SPL*, which describes the variability of the domain. For the measurements we needed to look not only at the compression software but also at the complete product including the input data. Considering the whole SPL we have many differences and commonalties leading to the possibility of first and second order interactions. To efficiently estimate the influence of such interactions common heuristics such as the pairwise interaction heuristic can be applied.

We used *FeatureHouse* [AKL12] to implement a generator for the *Measurement SPL*. The generator generates test data and compression programs for any sample. To configure, execute and control measurement runs we used *Apache Maven*[4] and the *Jenkins*[5] continuous integration server. This environment allows flexible and fully automatic measurement runs.

5.1 Methodology

To see whether considering input data in estimating a performance model can improve the precision of the estimated performance model we have to compare the precision of a performance model estimated with and without input data consideration.

For that we first estimated a performance model without considering input data. Second, we estimated a performance model considering input data. Finally, we applied both performance models on the three sets of configurations given by our heuristics and compared the

[4]http://maven.apache.org
[5]http://jenkins-ci.org

estimated non-functional properties (x) with the realized non-functional properties. These sets differ in their sizes and their ability to infer feature interactions. The deviation of the predicted value (p) to the realized value (r) is the *prediction error* (e)

$$e = |x_p - x_r|.$$

We use absolute differences, as they are easier to interpret and more robust for small values than relative differences. We compared the prediction errors of the performance model without input data (w/o) and the performance model with input data (w)

$$\Delta e = e_{w/o} - e_w.$$

A positive value means that the error of the performance model with input data is smaller than the performance model without input data. We tested the significance of this difference using the non-parametric Wilcoxon signed-rank test according to the test hypotheses:

H0 The prediction error of the model estimated with input data is equal or higher than the prediction error of the model estimated without input data.

H1 The prediction error of the model estimated with input data is less than the prediction error of the model estimated without input data.

We performed the measurement runs on a desktop PC with an Intel Core 2 Duo 3.0 GHz processor, 6 GB RAM and Windows 7 as operating system installed. Our Compressor SPL relies mostly on Java for which we used the Oracle Java Virtual Machine 1.7.

For this rather small case study it was possible to measure every configuration within a few hours on our experimental setup to create the performance models. However, as we will see later, good prediction can already be reached with a small subset of measured configurations. This aspect is important for the scalability of the proposed solution and is therefore explicitly addressed in our study.

We generated the performance models on the basis of three different sets of sample configurations. The *minimal* configuration set contains enough configurations to use regression analysis to estimate the impact of each feature. The *pairwise* configuration set extends the minimal configuration set to include additional configurations to observe interactions between two features. The *three-wise* configuration set extends the pairwise configuration set to observe interactions between three features.

Which configurations are part of the respective sets had to be calculated. Johansen et al. [JHF12] developed algorithms and tools to calculate the set of pairwise configurations which we used to build the set in our case study. The test of all possible three-wise interactions would require a high number of measurements. Hence, heuristics had to be applied to reduce the number of required configurations. We applied our three-wise heuristic described in Section 4 to calculate configurations in order to identify important second order interactions.

The measurement of compression ratio and memory usage is deterministic. Therefore, we were not faced by any threat of biased measurements for these non-functional properties.

However, the non-functional property compression time is vulnerable for biased measurement, which we tried to reduce by several strategies. First, we had a warm up phase, in which we ran the configuration twice without taking time to allow the virtual machine to apply optimizations. Then, we ran the test again four times and took the average runtime as our result. A look at the standard deviation of the four measurements showed that we got stable results. Second, we executed the performance measurements as the exclusive operation in the execution environment. Third, we set the memory of the virtual machine large enough to prevent disturbing effects from the Garbage Collector and executed all operations in memory so that disk or network I/O will also produce no disturbing effects.

5.2 Results

In the next section we present the results of our study separated by the measured non-functional properties. First, we shortly discuss the different sizes of the configuration sets, which are required to apply the discussed interaction detection approaches. Second, we report for all three considered non-functional properties the prediction errors $\overline{e}$ for both performance models and their differences $\overline{\Delta e}$.

The required sizes of the configuration sets are reported in Table 1. To evaluate the impact of every feature (including data features) a *minimal* set of 28 configurations is required. The *pairwise heuristic* already requires a total number of 150 measurements.

Table 1: Size of the configuration sets

Configuration set	Size	Relative size
Minimal	28	0.02
Pairwise	150	0.1
Three-wise (heuristic)	349 - 443	0.24 - 0.31
Three-wise (complete)	729	0.51
All	1440	1

The *three-wise heuristic* leads to 349 measurements for ratio (443 for time and 372 for memory), which equals 24% (31%, 26%) of all possible configurations. The configuration set of the three-wise heuristic is a superset of the pairwise set. It contains 199 to 293 *additional* configurations compared to the pairwise set. The difference between both sets cannot be generalized as it depends on the feature model as well as on the results of the pairwise measurements.

The three-wise heuristic configuration sets are specific for each non-functional property. The more non-functional properties are considered in a study the more configurations are required to identify three-wise interactions for the considered non-functional properties. However, the size of the sets does not grow linearly because the sets share a large amount of configurations (between 56% and 61% of the configurations).

As shown in Table 1 only 2% of all possible configurations are enough to create a performance model which considers direct effects of features on non-functional properties. The more interaction effects have to be captured the more configurations are required. In case of the three-wise heuristic nearly half of the possible configurations had to be measured. Compared to the fact that all possible configurations have to be considered without the use of this heuristic, this is still a good improvement.

Based on the generated configuration sets, we built performance models with and without considering variability in the input data. The results of our evaluation are summarized in the Tables 2 – 4.

Table 2: Regression for compression ratio

	Without input data		With input data			
Configuration set	$\overline{e}$	σ	$\overline{e}$	σ	$\overline{\Delta e}$	p_{wrt}
Minimal	0.91	0.50	0.09	0.25	0.82^{***}	<0.001
Pairwise	0.35	0.33	0.07	0.19	0.28^{***}	<0.001
Three-wise heuristic	0.30	0.35	0.05	0.15	0.25^{***}	<0.001

*** $p < 0.001$

Table 2 shows the results of the performance models for compression ratio. The prediction accuracy has greatly improved once data features were considered. Even with the minimal set the prediction quality can be improved to a much better but still not perfect average error of 0.09 compared to the average error of 0.91 without input data. The three-wise heuristic with input data gives the best results with an average error of 0.05 for the performance model with input data compared to an average error of 0.30 for the performance model without input data. For all configuration sets the study showed a significant improvement by considering input data.

Table 3: Regression for memory usage (in kB)

	Without input data		With input data			
Configuration set	$\overline{e}$	σ	$\overline{e}$	σ	$\overline{\Delta e}$	p_{wrt}
Minimal	507.2	2313.53	545.00	2366.67	-37.78	1
Pairwise	497.70	2057.33	47.53	183.27	450.17^{***}	<0.001
Three-wise heuristic	496.46	2052.99	47.43	188.22	449.03^{***}	<0.001

*** $p < 0.001$

Table 3 shows the prediction of the non-functional property memory usage. For the minimal configuration set the prediction with input data cannot be assumed to be better than without input data. However, the more information is provided by the considered sample the better the predictions of the model with input data get. This is not the case for the performance model without input data. Even with many more measurements the prediction error has only slightly improved. The model without input data is not able to consider a major influence on the dependent variable. For the larger configuration sets the precision model with input data is significantly more precise than the model without input data.

Table 4 provides a different image than the results of previous non-functional properties. In the minimal and pairwise configuration sets the performance model with input data leads to a higher prediction error than the simpler model without input data. The direct and pairwise interaction effects provide no good explanation of the observed values. The two smaller configuration sets provide no information of second order interactions. The direct and pairwise effects are overemphasized. Only if second order interactions are considered with the three-wise heuristic the model with input data provides results which are significantly better than the model without input data.

Table 4: Regression for compression time (in seconds)

	Without input data		With input data			
Configuration set	$\overline{e}$	σ	$\overline{e}$	σ	$\overline{\Delta e}$	p_{wrt}
Minimal	4.20	24.54	4.58	23.96	-0.38	1
Pairwise	6.42	23.91	12.57	38.41	-6.15	1
Three-wise heuristic	4.22	24.55	1.62	12.78	2.60^{***}	<0.001

$^{***}\ p < 0.001$

5.3 Discussion

For all three non-functional properties we can show significant improvements compared to respective models without input data. In case of compression time the performance model with input data needs a three-wise heuristic configuration set to achieve significantly better results. This suggests that some second order interactions between the features of the SPL have a large influence on compression time and have to be considered to achieve precise prediction results. Our study showed that the shape of the input data has a significant effect on non-functional properties of the Compressor SPL. Based on our study we can assume that in some domains contextual influences can have a noticeable effect on non-functional properties.

The measurement effort is higher if input data is considered as an additional source of variability than without it. Using heuristics in the selection of configurations mitigates this effect.

However, the results should be seen under the light of some possible threats to their validity. The measurement of performance metrics can be subject to a variety of errors. We only use these measurements for comparison and not the absolute values so that they should not affect our conclusion. The presented predictions still have errors. The SPL contains pairwise and second order interactions of which not all are covered by our prediction model. This has to be seen as a trade off between measurement effort and accuracy.

The used SPL was especially created for this study and is designed to make the interaction between data features and features highly visible. The selected data features represent extreme values within the range of possible realistic values. For example, completely randomized data on the one extreme and data of only zeros on the other extreme will show extreme behavior of the compression ratio. Nevertheless patterns of the modeled data will also appear in real data, yet, not in pure form but in combination with other patterns of data. This restricts the generalizability of our study but still gives a good indication that contextual influences can have an effect on non-functional properties.

Despite the fact that our case study is based on an artificial SPL we can see that input data can indeed have an effect on non-functional properties. The results of our study are somewhat extreme and cannot be extrapolated to arbitrary cases. However, we have shown that an effect can exist. To improve the generalizability of our findings the next step has to be a study with real-world SPLs.

6 Conclusion

With the example of input data we showed how contextual influences can be included into the measurement of non-functional properties of SPLs. We applied our approach in a synthetic case study from the domain of lossless compression algorithms to study whether input data has an effect on three instances of non-functional properties.

Our study shows that input data has an effect on the precision of measured performance models. We see this as an indicator that considering contextual influences can have a positive effect on the precision of performance models. This is a prerequisite for future work to validate the approach using realistic SPLs. A natural choice of study candidates are the publicly available and realistic SPLs described by Siegmund et al. [SKK$^+$12, SRK$^+$12].

To conclude, in our synthetic case study we found that contextual influences can improve prediction of non-functional properties. This is a first indication that the accuracy of prediction models can be improved if contextual influences are considered. It is therefore worthwhile to carry out further extensive case studies with real-world SPLs.

References

[AKL12] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering (TSE)*, 2012.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, USA, 2000.

[CGR$^+$12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 173–182, New York, NY, USA, 2012. ACM.

[CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a Feature: A Requirements Engineering Perspective. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering*, FASE'08/ETAPS'08, pages 16–30, Berlin and Heidelberg, 2008. Springer.

[CN07] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Boston, 6 edition, 2007.

[GCA$^+$12] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrezj Wasowski. Variability-Aware Performance Modeling: A Statistical Learning Approach. Tech. Rep. GSDLAB-TR 2012-08-18, Generative Software Development Laboratory, University of Waterloo, 2012.

[HBR$^+$10] Jens Happe, Steffen Becker, Christoph Rathfelder, Holger Friedrich, and Ralf H. Reussner. Parametric performance completions for model-driven performance prediction. *Performance Evaluation*, 67(8):694–716, 2010.

[JHF12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating t-wise Covering Arrays from Large Feature Models. In *Proceedings of the 16th International Software Product Line Conference (SPLC 2012)*, Salvador and Brazil, 2012. ACM.

[KR10] Lucia Kapova and Ralf Reussner. Application of Advanced Model-Driven Techniques in Performance Engineering. In Alessandro Aldini, Marco Bernardo, Luciano Bononi, and Vittorio Cortellessa, editors, *Computer Performance Engineering*, volume 6342 of *LNCS*, pages 17–36. Springer, Berlin and Heidelberg, 2010.

[POS$^+$11] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise Testing for Software Product Lines: A Comparison of Two Approaches. *Software Quality Journal*, 2011.

[Sal07] David Salomon. *Data Compression*. Springer, New York, NY, USA, 2007.

[SKK$^+$12] Norbert Siegmund, Sergiy Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting Performance via Automated Feature-Interaction Detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2012.

[SRK$^+$12] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3):487–517, 2012.

[SSPS10] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Approaching Non-functional Properties of Software Product Lines: Learning from Products. In *Proceedings of the Asia Pacific Software Engineering Conference*, APSEC 2010, pages 147–155. IEEE Computer Society, 2010.

[TP12] Rasha Tawhid and Dorina Petriu. User-friendly approach for handling performance parameters during predictive software performance engineering. In *Proceedings of the Third Joint WOSP/SIPEW International Conference on Performance Engineering*, ICPE '12, pages 109–120, New York, NY, USA, 2012. ACM.

A Pragmatic Approach For Debugging Parameter-Driven Software

Frank Ortmeier, Simon Struck, Jens Meinicke
{frank.ortmeier, simon.struck}@ovgu.de
Jochen Quante
jochen.quante@de.bosch.com

Abstract: Debugging a software system is a difficult and time consuming task. This is in particular true for control software in technical systems. Such software typically has a very long life cycle, has been programmed by engineers and not computer scientists, and has been extended numerous times to adapt to a changing environment and new technical products. As a consequence, the software is often not in an ideal condition. Additionally, such software often faces real-time requirements, which often makes it impossible to use dynamic techniques (e. g., single-stepping or tracing).

Technically, such software is often realized in C/C++ in a rather imperative programming style. Adaptation and extension is often done by adding configuration parameters. As a consequence, checking for correctness as well as debugging requires to consider the source code as well as its configuration parameters. In this paper, we propose a pragmatic approach to debugging such software. The approach was designed such that (a) it does not require any understanding of the software before starting, and (b) that it can be easily used by programmers and not only by experts. We evaluated the approach on an artificial but realistic case study provided by Robert Bosch GmbH.

1 Introduction

Over the last decades, computing power has risen dramatically. This lead to a completely new generation of software systems as well as new processes for software design and construction. Software development became professionalized – numerous, elaborate design processes and development tools made software construction very efficient. However, new processes and design methodologies often focus on information systems. They are very well tailored for (new) development of IT support for business processes of a company.

On the other hand, software is taking over more and more functionality in many technical applications. In a modern high-end car, the source code of the software now exceeds one million lines of code. Software development and maintenance sums up to almost 40% of the total development costs for a high-end car (according to a statement of a leading German automotive manufacturer).

Embedded software has now grown over more than three decades. As new versions are often implemented by adapting existing software controls, the set of parameters controlling the software has become very complex. In addition, documentation is often poor, and the programmers might even have retired. This makes maintenance, extension and debugging very expensive. For mid- and long-term improvements re-engineering such software systems (into modular and extensible architectures [SH04, KKLK05]) is a worthy goal. However, this requires quite some effort and time.

In this paper, we propose a pragmatic approach for locating bugs in parameter-driven software. It can be used during re-engineering (specifically during testing) as well as for locating bugs after the release. We specifically address the problem of locating bugs. To debug a software that heavily depends upon parameters, it is not enough to only inspect the source code – you also have to consider the used (fixed) parameters. Only a combined view of both will make debugging possible. The core idea of our approach for locating bugs is to make use of the fact that software often has been running successfully in many scenarios, and that variants of the software only differ in the values of the configuration parameters. This allows for a structured search during debugging. We explain and evaluate the approach on an industrial challenge supplied by Robert Bosch GmbH [BQ11]. We want to explicitly point out, that with this approach we do not vote against solid software engineering or re-engineering software for increasing maintainability. We fully support this idea. The presented approach is not meant as an alternative, but rather as a method which can be used during all stages of bug-fixing in parameter-driven software. This ranges from fixing bugs detected during development (maybe in conjunction with automatic builds/tests in a continous integration environment like JENKINS/HUDSON) as well as locating bugs after the software is released.

In the following section (Section 2), we give a brief introduction to parameter-driven software and explain the challenges. The proposed approach for debugging is explained in Section 3, followed by a description of the industrial challenge and an application of the approach to this example (Section 4). Related approaches are discussed in Section 5. A summary and an outlook is given in the final section 6.

2 Parameter-driven Software

In many technical domains – for example automotive industry or production automation – control software has grown over several years or even decades. Over the years, more and more functionality was added and/or existing control software has been adapted to new series of products. Typically, this calls for a modular software architecture. However, in industrial practice, evolution/adaptation is most often controlled by parameters. Software controls in technical domains usually rely on a large set of parameters for correct functioning with the concrete hardware and physical process that is to be controlled. Each set of parameters is designed for optimal results for a very specific type of product – for the product that the software shall control. A certain share of these parameters is usually not adjusted by the company that develops the software, but by a different company that integrates it into its own hardware. We call the software itself together with a set of values for all parameters a *variant* of a *parameter-driven software*.

For such software, the choice of parameters typically heavily influences the behavior of the whole control software. Some parameters might even be used to (de-)activate major parts of the code in a top level if-statement. Therefore, we call such a software "parameter-driven". A parameter driven software program allows the user to specify and modify specific operations by only changing parameter values. This allows for highly customizable software. Of course, this comes at a price: understanding the software does not only require understanding the source code, but also understanding the parameters and specific sets of parameter values for a given product. This also includes understanding dependencies between different parameters.

Note that a parameter-driven software system can also be understood as a *software product line* (SPL): parameters that are evaluated at runtime are one way of implementing variability. However, in this paper, we do not focus on reengineering the existing software to a maintainable software product line, but rather describe a pragmatic approach for systematically debugging such a software. The presented approach has only been applied to parameters that are treated by the compiler, but it can also be applied to parameters that are directly manipulated in the binary. It also seems possible to extend the approach such that parameters for pre-processors (for example those used in *#ifdef* statements) can also be treated. In contrast to the parameter-driven software, *#ifdef*s typically alter the source code. However, if *#ifdef* statements are only used in a structured, "disciplined" way (i. e., only to in-/exclude entire functions or type definitions or sequences of entire statements in the source code [LKA11]), they could also be realized by a fixed parameter and a normal *if* statement.

3 Parameter-driven Debugging

In the following, we describe a pragmatic approach for systematic debugging of a parameter driven software. We assume that the source code as well as a number of parameter configurations is given. We also assume that documentation, design specification and expert knowledge is no longer available – at least not to such an extent that debugging is a trivial task. We further assume that a bug has been observed in one specific variant of the software, and that there exists at least one variant that behaves as specified.

Debugging is the process of reducing errors in computer software. This process is usually split in two separate tasks. First, the affected code lines need to be identified. Then the source code must be modified in such a way that the error disappears. Fixing the affected lines of code strongly depends on the source code itself and possibly requires much more insight knowledge. In our approach we thus concentrate on the bug localization step. However, knowing a limited number of erroneous lines of code greatly assists the subsequent correction of the bug.

Locating an error is a difficult task that can be tackled with various methods. In the case of parameter-driven software we make use of the fact that the software has typically been running in numerous variants for a long time and the bug only appears for a newly created variant. The core idea is to compare parameter sets of variants that are working correctly with those of the faulty variant. This is comparable to searching for an error in a new version of a previously updated software by looking at the differences in the source code made during the update. This process is applicable even if the developer has only less knowledge of the source code.

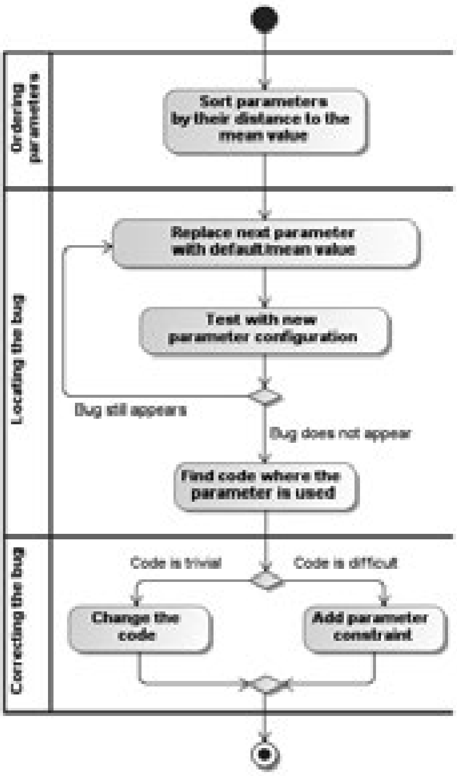

Ordering parameters
Sort parameters by their distance to the mean value
Locating the bug
Replace next parameter with default/mean value
Test with new parameter configuration
Bug still appears
Bug does not appear
Find code where the parameter is used
Correcting the bug
Code is trivial
Code is difficult
Change the code
Add parameter constraint

Figure 1: Overview of the approach for systematic debugging of parameter-driven software

Figure 1 shows the proposed approach. It can basically be split into three steps. In a first step, we try to identify candidates of possibly critical parameters. The good thing of parameter-driven software is that the source code itself is the same for all variants. So it is obvious that changes in the parameters are a good starting point for the search. The next step is to identify the "responsible" parameter(s) and the corresponding parts of the source code. Finally, the bug must be corrected.

3.1 Ordering parameters

In practice, a parameter-driven software will have hundreds or thousands of parameters. For a given variant and a given parameter, there most probably exists at least one other variant that differs in this parameter. Therefore, an effective search strategy is needed to identify the parameter(s) that cause the faulty behavior. A search strategy basically is an ordering on the set of parameters, which defines which parameter shall be analyzed first. In our approach, we want to present a generic concept. Thus, we may not rely on any application specific background knowledge.

An effective search strategy – if no background knowledge is available – is to start with the most "differing" parameter. There are different possible definitions of "differing". In this paper, we only use two intuitive definitions – (numerical) deviance from the mean value of the parameter and a quantile-based metric. The idea of the first one is that it makes sense to first look at parameters with a very large or very small absolute value (compared to the average). This is inspired by the fact that errors often occur if boundary values are used. The quantile-based "differing" metric basically follows the idea that if a parameter has only very rarely been used with a specific value, then it is more probable to cause an error than a parameter that is frequently used with the same value.

There may exist various other metrics like ordering by "runaway values" or even metrics that rely on domain-specific information. As each of them could be plugged in easily into the proposed approach, we do not go into more detail here. However, we found that even the two used, simple metrics allowed to quickly locate bugs in a third-party parameter-driven software of moderate complexity.

3.2 Locating the bug

After the identification of the most likely error causing parameters, we can substitute the parameters one by one with a default value and run the software. If the bug still appears, we proceed *incrementally* with the next parameter. Here, we follow the ordering of the parameters we obtained in the previous step. Typically, this will terminate at some point as we assume that the configuration of the software with all parameters set to default values behaves correctly.

Once the error disappears, we start the second step of the localization process[1]. Now that we have identified parameters that might be causing the error, we need to understand their effect. The first step is to find the place(s) in the source code where this parameter is used. Locating can be done easily with most IDEs. For example, in the Eclipse IDE, a function called "Call Hierarchy" solves this problem automatically. The "Call Hierarchy" view shows callers and callees for a selected Java member. It is also available for some other languages – in particular for C, which is interesting for the following real world example. The "Call Hierarchy" of a Field, Method or Class will show all places in the code that use it. Applied to the selected parameter, this will lead us to places in the source code where this parameter is used. Of course, the parameter could be used at numerous locations. However, as the software is parameter-driven, it is quite probable that the parameter is either only used a very limited number of times, or that the found locations may be re-

[1]Note that the debugging process depends on the order introduced by the concept "differing". Although it does not specifically address pairs of parameters, all such combinations are implicitly treated as if the parameters are set to default values *incrementally*. This means if the first parameter did not remove the bug, then we leave the first parameter at the default value and proceed with the second one.

duced significantly (e. g., by removing locations where it is only passed through from one function call to another).

Note that in theory, error localization might of course become hard if two parameters are far from each other according to the used metric *and* are used at very different locations in the code. However, it proved very efficient in practice as such situations seem to be rare. If such situations are common, the approach could be extended in various ways. We discuss such limitations and experiences after the case study.

3.3 Correcting the bug

The above steps reduced the whole source code to a few lines of code that are candidates for causing the error of the actual configuration. To fix the bug, one may now use any debugging strategy or code visualization technique of common software development you know. One well-suited technique for this step might be program slicing to find those pieces of the code that are influenced by the parameter.

Note that, after debugging the variant, in the worst case, some or even all other variants might have become faulty. So it is strongly suggested to (at least) run unit tests on some or all working variants. During bug fixing, the programmer usually focuses on the (single) not working variant and might easily forget about all the other perfectly working variants. So *before* fixing, the programmer must be made explicitly aware of this situation. A combination with documentation of changes and reasons for adaptation is also strongly suggested but not in the scope of this paper.

4 Case Study

This section describes the application of the proposed approach on a small system and scenario that was provided by Bosch and Microsoft Research as an "industrial program comprehension challenge" for ICPC 2011 [BQ11]. The team from University of Magdeburg successfully participated in this challenge, using the technique that is presented in this paper.

4.1 An Industrial Challenge

The task for the participants of the ICPC challenge was to find and fix a number of bugs in a robot leg control software. The (artificial) code contained typical elements of embedded control software, such as filters, ramps, and curves. It also showed the typical high variability that makes embedded software adaptable to different variants of hardware settings and customer requirements without changing the code. This was in particular reflected by a high number of application parameters.

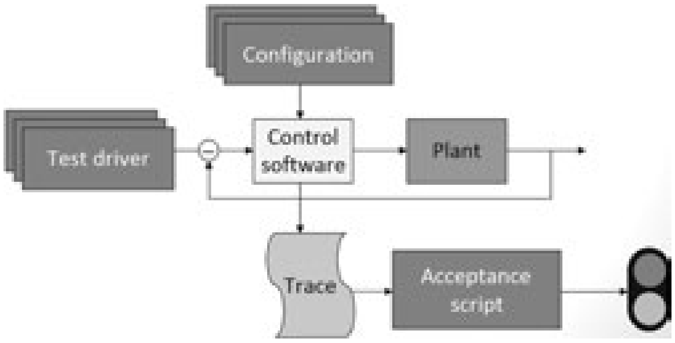


Figure 2: Test environment setup, as provided to the participants.

The core functionality of the software was to move a robot leg to a user-requested position. Certain limitations of the engine had to be obeyed for optimal performance of the leg. The software worked in general, but there were a number of bugs in the code. In particular, the following three bug reports were sent in by different (fictitious) customers:

1. The robot leg sometimes moves too slowly. In the example test case, one particular movement takes about 70 seconds, where it should only take 10 seconds. Other movements in the same test case are fast enough.
2. When the robot leg reaches its target position, it jiggles around instead of stopping.
3. Sometimes when it is moving, the robot leg explodes, sending shrapnel everywhere. After destroying 3 or 4 robots, the engineers believe the cause is due to the voltage that is sent to the engine being inverted too quickly. Also, sometimes the engine voltage drops much too quickly.

The participants got the following artefacts to work on[2]:

- The robot leg controller code (about 300 lines of C code, containing about 35 application parameters),
- a test environment, consisting of a simulation of the robot leg and a test driver (about 600 lines of C code),
- three bug reports, along with the concrete parameter configurations and test cases that show the erroneous behaviour,
- documentation (i. e., description of general idea and application parameters), and
- an acceptance test script to validate correct controller output.

For this challenge, the parameters were provided in separate header files. In reality, these parameters are usually directly changed in the binary (using special editors).

Figure 2 shows the overall setup of the test environment. The control software is only complete in combination with one of the configuration parameter sets. Different test drivers

[2]The complete challenge description and material is available at `http://icpc2011.cs.usask.ca/conf_site/IndustrialTrack.html`, accessed on 28 November 2012

simulate different scenarios, and the "plant" simulates the behaviour of the real-world robot leg. Finally, an acceptance script decides whether a test run was successful or not. In summary, we have a complete automated test environment for the robot leg controller software.

4.2 Applying the Approach

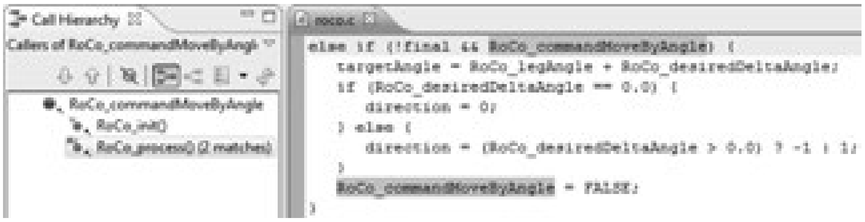
In this section, we describe the application of our approach to the ICPC industrial case study. Only one of the four variants of the product that were provided by the challenge was probably correct. The other three variants where faulty. As we will explain below, we were able to locate and fix all three bugs with our approach.

As we described above, we assume that no or only little knowledge or understanding of the code is available – and in fact, we (University Magdeburg) had absolutely no understanding of the code in the challenge when we started.

We start with step one of the approach, which is to find outliers in the parameters' values. Note that we did not implement an automatic ordering in the prototype, but rely on visualizing the metric to the programmer. This semi-automatic approach helps a lot if one is not sure which metric to use first. Technically, automatization would be straight forward. However, choosing the right metric, or even deciding how long to proceed with one metric and when to use another metric, is an interesting question.

4.2.1 Locating and fixing error 1

When comparing the parameter configuration of this variant to the configuration of the correct variant (and also to the other variants that showed different bugs), we found that the Boolean parameter *RoCo_commandMoveByAngle* was only used in the faulty variant (quantile-based metric). All other three variants used this parameter with a value "0", while in this variant, it was defined as "1". Changing the parameter to the default value "0" made the bug disappear. Examining the code – i. e., applying the *Call hierarchy* method – found only the single code sequence shown in Figure 3 using the parameter.


Call Hierarchy
Callers of RoCo_commandMoveByAngle
RoCo_commandMoveByAngle
RoCo_init()
RoCo_process() (2 matches)
roco.c

```
else if (!final && RoCo_commandMoveByAngle) {
    targetAngle = RoCo_legAngle + RoCo_desiredDeltaAngle;
    if (RoCo_desiredDeltaAngle == 0.0) {
        direction = 0;
    } else {
        direction = (RoCo_desiredDeltaAngle > 0.0) ? -1 : 1;
    }
    RoCo_commandMoveByAngle = FALSE;
}
```

Figure 3: The snipped of code which causes the first error

In this code snippet, we see a frequently used communication pattern in embedded software. To satisfy real-time constraints, embedded software is usually organized in time slices and uses a blackboard architecture [SCTQ09]. This means that messages between components – in particular in different time slices – are realized by writing to and reading from global variables. In the example, different types of commands are controlled by different values of variables. More concretely, the Boolean parameters control whether the robot leg will move to an absolute position or move relatively to its current position.

In order to fix the error, one still has to understand the source code. However, this becomes much easier now. For example, breakpoints or debug output can be introduced at exactly this part of the code (or even at all locations where the parameter is used). In this example, it is relatively easy to see that the assignments to the *direction* variable is wrong.

4.2.2 Locating and fixing error 2

For the next error, the quantile-based ordering of parameters did not show any significant parameters. Therefore, we chose to use the "deviance-to-mean" metrics.

Please note that two different metrics can be beneficial for visual analysis. One can either use the metric

$$\frac{\text{parameter-value}}{\text{mean-of-parameter-value}} \tag{1}$$

or its reciprocal. The first one is the standard, where one would plot the percentages of deviation to the mean. So if one parameter has a relatively high value, it will result in a peak in this plot. However, if a parameter has a very low (absolute) value, it will only be plotted around 0 and might thus be overseen. If the second metric is used, then both situations are switched: small values will become peaks and large ones will be plotted around 0. In this example, the second metric is more useful, as obviously very small values are used in this variant.

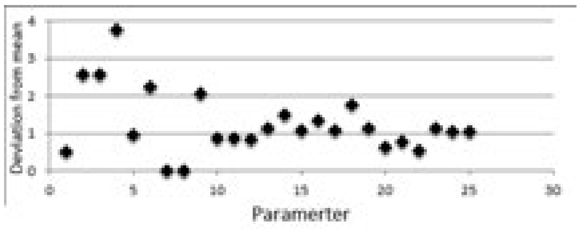
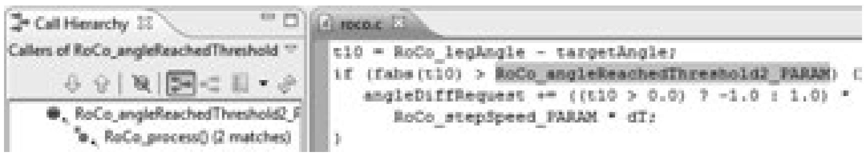


Figure 4: Deviation of parameters of variant 2 from mean values. The horizontal axis shows the 25 parameters. The vertical axis shows the corresponding result of the metric for small values.

In Figure 4, the deviation of parameter values of variant 2 are shown[3]. It does not contain Boolean parameters and parameters representing curves, because the metric needs numerical values. So there are only 25 parameters left in the diagram.

[3]Note that we did not include the Boolean parameters in Figure 4, as this metric does not make sense for them. Therefore, only 25 of 35 parameters are shown.

We see that the fourth parameter *RoCo_angleReachedThreshold2_PARAM* differs the most from its average value. It is almost four times smaller compared to the average. We start with this parameter and replace it with a default value. Now, we see that the error does not appear anymore. This means that the error is caused by this parameter. We then search for the usage of this parameter in the source code, which leads us to the lines of code shown in Figure 5.


Call Hierarchy
Callers of RoCo_angleReachedThreshold
RoCo_angleReachedThreshold2_P
RoCo_process() (2 matches)
roco.c

```
t10 = RoCo_legAngle - targetAngle;
if (fabs(t10) > RoCo_angleReachedThreshold2_PARAM) {
   angleDiffRequest += ((t10 > 0.0) ? -1.0 : 1.0) *
      RoCo_stepSpeed_PARAM * dT;
}
```

Figure 5: Code causing error 2.

Fixing the error requires understanding the code again. Here, the error is caused by an invalid combination of the threshold and step speed parameters. Considering the error description "jiggling" we decided to fix the bug, by implementing an adaptive dampening depending on the *RoCo_angleReachedThreshold2_PARAM* value.

4.2.3 Locating and fixing error 3

For finding the cause of the third error, we look at the deviation from mean values again (see Figure 6).

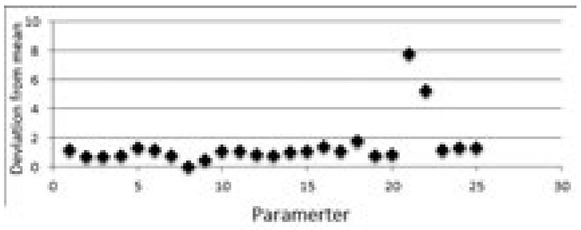


Figure 6: Deviation of parameters of variant 3 from mean values.

It is easy to see that two parameters – 21 and 22 – differ a lot from their average: they have very low absolute values. These parameters are called *RoCo_TimeSlopeNeg_PARAM* and *RoCo_TimeSlopePos_PARAM*. If we replace their values by default ones, we see that the voltage no longer drops rapidly, and the bug disappears. This means that these parameters are responsible for the faulty behaviour. The "Call Hierarchy" applied to both parameters leads us to the call of the method *Ramp_out()* in the *RoCo_process()* method.

The affected lines of code include relatively difficult mathematics (which is used for calculating velocity profiles). It is now again an application-specific task to correct this bug. If the mathematics may not be correctly re-designed such that they can deal with the given parameters, then the it should be explicitly noted what acceptable ranges for parameters are[4]. For example, this could be made visible in the parameter description by a note and could also be integrated into the source code by range checking at runtime, which might for example trigger an exception indicating that the parameter has been used with incorrect values[5].

4.3 Experiences

While applying the approach to the industrial challenge, we were surprised how easy it was to locate the source code snippets that were responsible for the bug (even for us who had no understanding of the code before). Although we only used relatively simple metrics, they quickly led us to the bugs. This is even more interesting as only a very limited number of parameter sets was available. We did all the computations on the parameters in an Excel worksheet. However, it would be easy to integrate this functionality as a plug-in to Eclipse or Visual Studio. We also found that – theoretically possible – complex combinations of parameters values that are scattered throughout the code are rare. Currently, we are examining in another case study if this hypothesis also holds in larger applications.

Note, that the presented approach is about *locating* bugs in parameter-driven software. *Correcting* the bug is still a difficult task that cannot be solved uniformly. It also requires a good amount of program understanding that still has to be accomplished manually or by using adequate tools (such as a slicer). The advantage of the approach is that it quickly points the developer to those spots in the code that might be responsible for the bug. This reduces the overall required effort significantly.

Although we used this approach in an a posteriori scenario, where variants already existed and were deployed, we think that it will be of equal use during (re-)engineering of such applications. In particular a combination with highly iterative builds in a continous integration environment might be promising.

5 Related Work

In this section we first take a broader look at alternative approaches. After that the other submissions to the ICPC 2011 challenge are discussed.

A feature model depicts the hierarchy and relation between features of a software product line in a formal way [Bat05]. Thus, there is a lot of research on debugging product lines based on the feature models. For example Batory as well as Benavides et al. propose the usage of a constraint satisfaction problem to decide between valid and invalid variants with respect to the feature model [Bat05, BMAC05]. White et al. also continue this approach by determining the most minimalistic changes to a given invalid configuration to solve all conflicts [WSB$^+$08]. Of course, these techniques perform well if there is a feature model

[4]Of course, such situations must not happen and therefore we should note it explictly in the specification
[5]This could of course also be combined with other source code annotation methods like SPEC# or JML.

and/or a precise specification of the product line. Due to the missing of a specification for the case study used in this paper their approaches are not applicable here. The absence of a feature model is not uncommon. Either a simple software product evolved to a product line over the time and thus has no product-line specific design. Or under strong time-to-marked requirements there simply might not be enough time or motivation to develop a complete feature model. Our approach performed well even without further specifications or domain specific knowledge.

Testing is an well known technique to detect failures before deploying a product. This holds also for software product lines. We believe that an elaborated test process could point out at least some of the demonstrated failures. But as well as for the feature model based debugging approaches the process of testing starts in a very early stage of the software design [PM06, MvdH03]. In addition, testing also requires detailed knowledge of the domain and/or the implementation. Thus, locating the bugs in this case study based on an elaborated test process was not feasible.

Statistical analysis of the run-time behaviour of a program can also lead to the buggy core regions [LYF$^+$05, CLM$^+$09]. A clear benefit of these approaches is the assumption that there is no further knowledge about the semantics of the program. The statistical data about the program execution is collected via program instrumentation. Due to limited resources (memory and computational) or strict real-time requirements in embedded systems the insertion of additional code might be impossible. Also these approaches do not exploit the information that the bugs might depend on certain parameter configurations.

In the original ICPC challenge [BQ11] there were four other submissions that applied different techniques to identify and fix the bugs:

- A Frama-C[6] based approach, using interpretation, slicing, and tracing. This approach is technically very complex, requires a strong infrastructure and good knowledge of the tool, but identified all the faults in the code. It also provided a good level of program understanding.
- Control and data flow visualization based approach. This one provided the weakest results, probably due to wrong interpretation of the extracted flow graphs.
- Spectrum-based fault localization [AZvG07] (two submissions). This technique identified some of the bugs, but completely missed others.

In comparison to our approach, all other submissions either require an advanced infrastructure and a high level of knowledge about it (Frama-C), or they were much less successful in the challenge. Our approach is by far the most lightweight and easy to learn, and therefore also the one that is closest to applicability in daily practice. Additionally it provides a well-balanced combination of automation and manual understanding by pointing to parts of the code that one should look at in more detail without causing much effort.

[6]http://frama-c.com/

6 Conclusion and Outlook

We presented a lightweight approach for locating bugs in parameter-driven software. The approach is designed according to two simple requirements from industrial practice: (i) It should be easy to apply, and (ii) it should be as generic as possible. We achieved this by structuring the debugging process into two steps. First the the responsible parameter is identified, and then the locations in the source code are identified. To meet both requirements, we only rely on standard functionality that is available in most IDEs and a simple methodology for locating the responsible parameter. As a consequence, the approach can be applied in many domains by practitioners without much additional training or software. This distinguishes the approach from most others. However, the approach does not provide a recipe for fixing the code. This part still requires a good level of understanding of the program. Manually gaining such an understanding is usually very expensive. Therefore, an additional strategy or tool is needed to support this process. One probably well-suited approach is program slicing, which can indicate those parts of the code that are influenced by an identified parameter.

We successfully applied our approach to an industrial case study provided by Robert Bosch GmbH. The case study was presented as an industrial challenge on the International Conference on Program Comprehension 2011. It is important to note that the case study firstly stems from an industrial practitioner and secondly was defined completely independent of the presented approach. Nevertheless, we where able to quickly locate and fix three different bugs. Efficiency and practicability was judged by the evaluation board of the challenge to be very good. Only a single other approach was rated better (Frama-C), but this one relies on very elaborate static analysis methods. It is further worth to note that the complete analysis was done by a bachelor student who had no a priori experience in parameter-driven software. Therefore, we believe that the approach is easily usable in industrial practice. In addition, it is easy to adapt the approach to specific needs of various domains or companies. This ranges from domain specific orderings and/or integration of automatic tools for calculating the orderings.

An extension of the proposed approach could be a combination with a dynamic coverage differencing approach (similar to the feature location technique by Wong et al. [WGHT99]): the program would be executed twice, first with the faulty configuration, and after that with the configuration where only the faulty parameter is changed. In the next step, the executed lines of the two runs are compared to each other. This could help to identify parts of the code that are only executed with certain parameter settings. Similarly, a combination with tools for static analysis (which can for example extract control flow graphs) seems promising.

In future work, we will investigate how good the presented orderings work for other parameter driven programs, and if the approach can be applied to other types of software (e. g., software with many *#ifdef* statements) as well. In particular, we also plan to evaluate it on a large scale software system from the automotive industry.

References

[AZvG07] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07)*, pages 89–98, 2007.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Berlin / Heidelberg, 2005.

[BMAC05] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 381–390. Springer Berlin / Heidelberg, 2005.

[BQ11] Andrew Begel and Jochen Quante. Industrial Program Comprehension Challenge 2011: Archeology and Anthropology of Embedded Control Systems. In *Proc. of 19th Int'l Conf. on Program Comprehension (ICPC)*, pages 227–229, 2011.

[CLM$^+$09] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Proc. of the 31st International Conference on Software Engineering (ICSE)*, pages 34–44, 2009.

[KKLK05] K.C. Kang, M. Kim, J. Lee, and B. Kim. Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets–a Case Study. In *Proc. of 9th International Software product lines conference (SPLC)*, pages 45–56, 2005.

[LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM Press, March 2011.

[LYF$^+$05] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[MvdH03] H. Muccini and A. van der Hoek. Towards testing product line architectures. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 82(6):99–109, 2003.

[PM06] Klaus Pohl and Andreas Metzger. Software product line testing. *Communications of the ACM (CACM)*, 49:78–81, December 2006.

[SCTQ09] Vincent Schulte-Coerne, Andreas Thums, and Jochen Quante. Challenges in Reengineering Automotive Software. In *Proc. of 13th Conf. on Software Maintenance and Reengineering (CSMR)*, pages 315–316, 2009.

[SH04] M. Staples and D. Hill. Experiences adopting software product line development without a product line architecture. In *Proc. of 11th Asia-Pacific Software Engineering Conference (APSEC)*, pages 176–183, 2004.

[WGHT99] W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan, and Kishor S. Trivedi. Locating Program Features using Execution Slices. In *Proc. of the IEEE Symposium on Application-Specific Systems and Software Engineering & Technology*, pages 194–203, 1999.

[WSB$^+$08] J. White, D.C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Proc. of Intl. Software Product Line Conference (SPLC)*, pages 225–234, 2008.

Software in the City: Visual Guidance Through Large Scale Software Projects

Marc Schreiber
FH Aachen
University of Applied Sciences
marc.schreiber@fh-aachen.de

Stefan Hirtbach
Microsoft
Advanced Technology Labs Europe
stefah@microsoft.com

Bodo Kraft
FH Aachen
University of Applied Sciences
kraft@fh-aachen.de

Andreas Steinmetzler
Microsoft
Office:mac
andreast@microsoft.com

Abstract: The size of software projects at Microsoft are constantly increasing. This leads to the problem that developers and managers at Microsoft have trouble to comprehend and overview their own projects in detail. Regarding that there are some research projects at Microsoft with the goal to facilitate analyses on software projects. Those projects provide databases with metadata of the development process which developers, managers, and researchers can use. As an example, the data can be used for recommendation systems and bug analyses.

In the research field of visualization software there are a lot of approaches that try to visualize large software projects. One approach which seems to reach that goal is the visualization of software with real life metaphors. This paper combines existing research projects at Microsoft with a visualization approach that uses the city metaphor. The goal is to guide managers and developers in their day to day development decisions and to improve the comprehension of their software projects.

1 Introduction

Microsoft's software projects are generating a vast amount of data during their development process. Their Source Code Control Systems (SCCSs) alone are several terabyte large, in addition information is stored in bug repositories, drop shares, etc. With respect to that it is nearly impossible to keep a clear overview of the software project in various ways.

The pure size of the SCCSs makes it difficult for the single developer to understand the coherence of the source files, classes, functions, and other software components. Another example is that it is hard to tell which components (libraries and executables) rely on each other due to countless number of components inside of each Microsoft project. Those projects contain legacy code as well, which makes it harder to understand the code base, especially when the original author of legacy code left the company.

Within Microsoft Research there are several projects trying to help developers to manage the large scale projects. An example is the defect predictor of [NZZ$^+$10] which predicts code defects inside of Windows. The predictor identifies defects by certain consecutive changes on code in the SCCS history. The predictor claims to be the best with precision and recall above 90%.

Codebook is a social network based approach of [BKZ10] which helps Microsoft developers to orientate in large scale software projects. It is a platform which models the relationships among people, bugs, code, tests, builds, and specifications.

Those examples have in common that they rely on the data of the SCCS—there are more examples which also depend on other data like bugs: [AJL$^+$09], [BN12], [ZWDZ04], etc. The common set of requirements from those projects motivated the creation of a new data warehouse inside of Microsoft facilitating analyses on top of development processes. This data warehouse consists of multiple databases:

Core Database (CDB) Contains raw meta data information of SCCSs like Changelists, Filechanges, differences between files, etc.

Derived Database (DDB) Stores heuristic data which is generated on top of CDB data.

Binary Database (BDB) Saves information about binaries (types, functions, etc.).

Task Database (TDB) Includes data about workitems, bugs, and their states.

With the available metadata of the development process the Product Groups (PGs) at Microsoft started to generate visualizations of their software. These visualizations are simple and provide only hints on what is happening during the development. At first glance the simplification of the visualizations seems to confirms the hypothesis of [Bro87]; the author argues that software can not be visualized because large software projects are too complex.

But [KM99] contradicted [Bro87] and showed that *"Visualization is Possible"* if metaphors of the real life are considered. Many software visualizations are successful when city metaphors are used. Their advantages will be described briefly in Section 2 by demonstrating some existing approaches for visualizing software with the city metaphor.

Inspired by the existing visualizations, Section 3 will introduce the concepts of a new visualization tool named CodeMine City Tool (CMCT). This tool visualizes large software projects on top of the data warehouse with the goal to guide developers and managers in their day to day decisions and to make the comprehension of the software projects better.

Section 4 will show some results of visualized Microsoft projects. Following this Section 5 will describe some future prospects of the CMCT and Section 6 will give a conclusion talking about lessons learned and takeaways for developers.

2 State of the Art

In the research field of software visualization the consideration of the city metaphor claims to be successful—examples are the work of [KM00], [BNDL04], [WL07], and [SL10].

They are all using metaphors of the real life; [KM00], [WL07], and [SL10] use the city metaphor and [BNDL04] uses the landscape metaphor. Their visualizations are using similar concepts, for example, they are all using buildings to represent elements of software.

In [KM00] the authors developed a tool called *Software World* which visualizes Java code with the city metaphor. The tool was a result of their previous paper [KM99] where they showed that the visualization of software can be meaningful even when the size of software projects prevents a two-dimensional visualization (graphs, UML, etc).

Software World displays a whole software system with multiple cities where each city represents one source file. The cities are subdivided into districts representing classes of the source files and the buildings in the districts visualize the methods of the classes. The building height is defined by the number of source code lines and the building color indicates if the access is public or private.

The paper [WL07] introduced a similar concept to [KM00] for Java software visualization: A city represents a whole Java project, the districts represent Java packages, and the buildings represent Java classes. The dimensions of the buildings (area, height) are determined by the number of attributes respectively number of methods of each class.

An innovation of [WL07] is the introduction of the district topology which displays the Java package hierarchy. Each district is located on a level according to its nesting in the package hierarchy. A subdistrict of a district is placed one level higher. This topology creates a hilly landscape which helps to realize the package structure.

The authors of [BNDL04] focused on the structure of software projects. In contrast to [KM00] and [WL07] buildings are used to symbolize the existence of attributes and methods distinct by different shapes. Another difference is that no districts are used to group classes. Instead the authors used nested spheres to symbolize the package structure. So inside of a sphere there are other spheres (the subpackages of a package) or cities where each city represents one class and the buildings inside the attributes and methods of that class.

An important aspect of [BNDL04] is to display the dependencies among attributes and/or methods. Rather then using straight lines for direct connections, which could cause occlusions and overlappings, the authors propose the *Hierarchical Net*. This solution routes the connections between depending attributes/methods according to the hierarchy levels of software elements.

[SL10][1] uses the city metaphor as well by focusing on visualizing the development history. Like the other introduced papers [SL10] takes advantage of the hierarchical structure of software represented by a hierarchical system of streets. For visualizing the history of software they use a layout algorithm which places components along the streets but keeps their initial positions: New components are attached to the end of a street, growing components cause to shift other elements, and the space of removed component stay empty.

A second approach for visualizing the history is the introduction of a topology to the system. An important aspect of the development history is the age of components which is represented by the level of each component. *"The older an element is, the more its*

[1] [LS09] presents the approach as well.

representation will be elevated in the visualization.”[2] For a better age comparison of different heights in the topology they also used contour lines. This concept makes it is for the user to tell how old components of the software are.

3 Concept of CodeMine City

The basic idea of the CMCT is to provide managers and developers of software projects with a visualization which guides them in their development decisions. Another purpose of the tool is to help managers and developers to comprehend their software better, e. g. by visualizing library dependencies.

As seen in Section 2 three dimensional visualizations which use a metaphor of the real world—especially city metaphors—can help to gather the vast amount of data of software projects. So to guide users of CMCT in their decisions and to help them comprehending their software better the tool will visualize data in a city metaphor, like in [WL07, page 2]. But the tool will have three main differences:

(A) Instead of visualizing classes as buildings the CMCT will visualize binaries as buildings.

(B) The CMCT will characterize its cityspace by the dependency graph of the binaries.

(C) The user can decide which metrics will determine the dimensions of the buildings like area or height.

The reason for visualizing binaries as buildings (point A) is the scale of projects in Microsoft. If classes would from the basis for buildings the cityspace would explode. The number of buildings in this approach would cause quite a bit of confusion.

	Small scale project	Large scale project
Binaries	6×10^1	3.6×10^2
Source Files	5.2×10^2	5.7×10^4
Classes	8.2×10^2	9.6×10^4

Table 1: Compare Buildings Count of Microsoft Products on the Basis of Binaries, Source Files, and Classes

As Table 1 shows the amount of buildings for a large scale project would be about 160 times bigger if the cityspace would be based on the source files. Even in a small scale project the city would be about ten times bigger. If the buildings were based on classes, it would be even worse. This indicates that the traditional approaches as seen in Section 2 will lead to confusing cityspaces.

Those approaches could also not be applied if a project is based on C code alone because the buildings must be based on functions. But what would happen when a project is based

[2]Source: [SL10]

on C and C++? Those arguments underline the decision for point A because it is generic approach for software projects in general.

The main argument for point B is that the user can test applied changes to a binary to understand the affect on other binaries in the system. This point is very important in large scale software projects because inter binary dependencies are often non-obvious. For example: A new developer in a team does not know the structure of the software project. If he has to fix bugs he can determine with the tool if that fix could harm other binaries and he can ask his team mates for advice.

A second reason for point B is that on the basis of the cityspace it is also possible to identify the role of a binary: Binaries at the center (core binaries) of the city are referenced more often within the software project, whereas peripheral binaries are referenced less because they make use of the core binaries.

The visualizations in [WL07] and [KM00] do not let the user choose between different metrics[3]. That is a disadvantage because decisions are rarely based on a single metric like lines of code which supports point C. The CMCT allows to select different metrics[4] which makes it possible to detect binaries with the help of their shape or color. Section 3.2 will explain the details of that argument why point C is important and how shape and color can help.

3.1 Setting of CMCT Cityspace

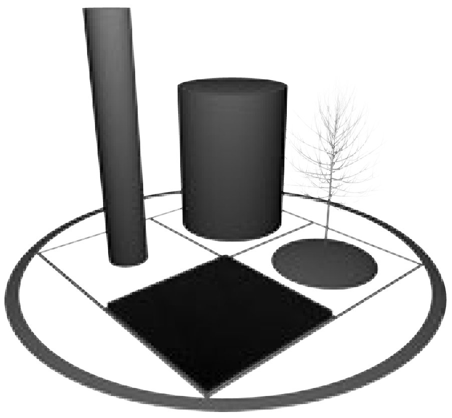


Figure 1: Elements of CMCT

The cityspace of the CMCT consists of four different elements (see Figure 1). As seen at the beginning of Section 3 binaries will be visualized by buildings and the dimensions are determined by metrics which the user is able to choose. If no metrics could be applied to a binary, because no source files are available for the binary (external binary), then the binary will be symbolized by a tree. This makes it easy for the user to distinguish between internal and external binaries.

The binaries are organized in districts surrounded by a roundabout traffic and each district contains the binaries of one directory. In the district the buildings are arranged on a grid structure. If a districts contains n buildings and n is not a square number, then some carets are empty. From the bird's eye view it is hard to differentiate if a caret is really empty or if it contains a building with a very small area. Meadows filling empty spaces and help to differentiate these cases.

[3]In [BNDL04] there is not even one metric displayed.

[4]The metrics work the data warehouse which mean that they are using data from CDB, DDB, BDB, and TDB.

One concept of the CMCT is to visualize the dependencies among the districts. This is supported by streets connecting dependent districts with each other. Single binary dependencies can be resolved through the selection feature of CMCT: The selected binary will be highlighted with blue color, all binaries which reference the selected one will be highlighted with orange color, and all binaries which are referenced by the selected one will be highlighted with green color. In addition the CMCT will show the dependency route with orange or green lines on the connecting streets.

The arrangement of the cityspace or more precisely of the districts happens through two algorithms: At first the districts are arranged by the algorithm described in [KK89] and afterwards overlappings of the districts are removed with the algorithm of [DMS05].

A property of the algorithm in [KK89] is that it produces a relative small number of edge and node crossings. However crossings are not eliminated and because of this fact the CMCT has to avoid that districts are crossed by streets[5]—streets can be crossed by other streets—which is supported by tunnels passing underneath the districts.

The tunnels help the user to identify the dependencies in an easier way. If the crossing of district would be avoided by the usage of the roundabout traffic (crossing streets enters the roundabout traffic on one side and leaves it on the opposite side), it could create the assumption that two districts depend on each other even if they do not.

3.2 User Stories

Clean and Organize Code in Projects Over time the source code of projects grows and grows. Often code will be refactored or functions become obsolete resulting in dead code. To help developers cleaning up the code base the CMCT can display a texture with cracks for each building. The more cracks are visible the more dead code the building contains.

The CMCT can also help to organize libraries. If there are a lot of functions which are only called by one or a few other libraries, it makes no sense to provide those functions in the core library. A better solution is to make those functions part of a new library or to move them into existing ones. Therefor the CMCT can display, with a pie chart texture, how much code is used by none, few, or many other binaries.

What Depends on a Library/Source File? A new developer at the team was told to implement a new feature. For that he has to work on code which he has never touched and so far he does not know which other binaries could be impacted by his changes.

With the CMCT he can search for the file which he will modify and the CMCT will directly show all binaries depending on it. If no dependencies exists, he can go on implementing the feature. Otherwise he can investigate who is working on dependent binaries and coordinate the work

[5]Crossing of districts would cause occlusions and overlappings, see [BNDL04].

What Files Are Easy To Edit and Which Are Not According to [SZZ05] it is possible to calculate how difficult it is to edit source files in a software project. This metric is interesting for managers of software projects because they can assign work to rookies or the experienced developers with respect to the difficulty of source files.

Combination of Metrics A wide variety of combinations of metrics for the different dimensions are conceivable. Here is one example: For the building area the user can choose the number of code lines and for the building height the number of methods. Buildings which appear as flat slices indicate that there is are lot of long methods in it—it is bad smell, see [Fow99].

3.3 Status of Implementation

The basic concepts of the CMCT are fully implemented and the tool is ready to use. The tool is configurable so that it can be used for all Microsoft projects whose data is available in the data warehouse. The CMCT provides a set of metrics to the user and the implementation is open to be extended with more metrics. The CMCT is implemented with C# and the rendering of the cityspace is done by WPF 3D (see [Pet07] as reference).

The CMCT is an internal project and will not be available outside of Microsoft. Part of the reason for this is its dependency on the internal data warehouse (CDB, DDB, etc.) tailored to Microsoft specific requirements.

4 Cityspaces of Microsoft Projects

Finally this section shows some cityspaces of software projects in Microsoft. The first example in Figure 2 is a large scale project showing a structured cityspace. In spite of the large number of binaries the cityspace is well-arranged. (1) The core district, on which a lot of binaries of the software project depend, is located at the center. Furthermore there is also a second core district near the center of the city. This district contains only one external binary on which almost every binary depends.

The districts which surround the core districts depend on them. (2) At the edge of the city there are some smaller groups of districts with interdependencies. All of those districts are co-located.

The highlighting of the dependencies works very well too. Through the selection it is possible to detect interrelated buildings immediately. Figure 2 also illustrates that the highlighting confirms the role of the district at the center as a core district.

All trees and meadows contribute to a good overview for the user, even from a far distance. By looking at Figure 2 it is very clear which carets of the districts are empty and which binaries are external to the project. It is also clear that most binaries are implemented by the project itself given the lack of trees in the city.

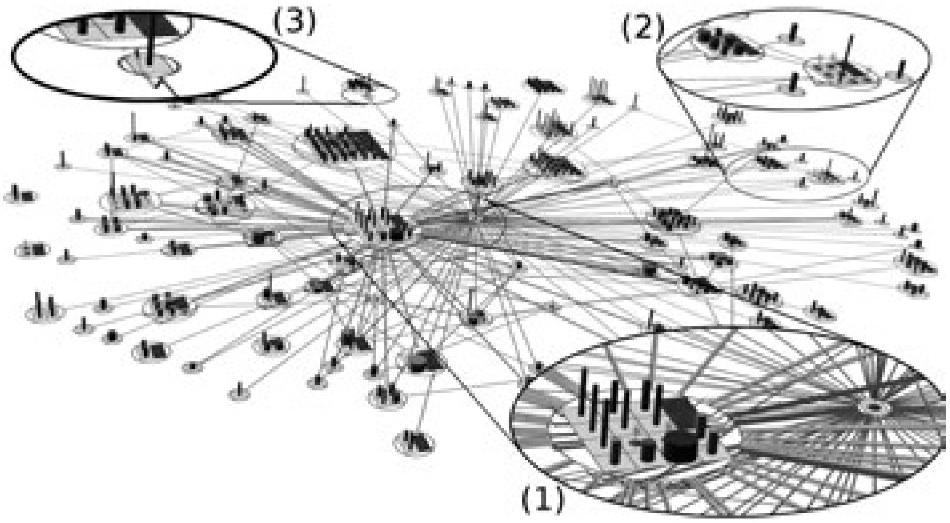


Figure 2: Cityspace of Large Scale Project

(3) From the distance it also possible to see if a district is passed by a tunnel. This and the fact that the route of a dependency between two districts never turns off the road makes it easy for the user to recognize if two districts depends on each other.

Figure 3 presents the cityspace of a small scale project. The cityspace of the project is well-arranged and it has one core district. There is a second core district but it is only referenced by the other core district, so they are forming the city center—the second core district looks like a park because it contains only external binaries and an open area.

The last example shows a cityspace of a medium scale project of Microsoft (see Figure 4). The city contains only of a few districts, where two of them contain 87% of all buildings. The remaining districts contain only a few buildings. Figure 4 shows that a lot of the binaries are external binaries symbolized by trees—an ecologically beneficial city.

The large district at the center contains 71 binaries. Its location and the amount of buildings in it create the impression that this district is the city center, but appearances are deceiving. The district contains an external binary on which almost all internal binaries depend. There is only one binary referenced outside of the district. The real city center is the district on left side of the big district. But that situation was quickly revealed with a few clicks and shows the power of CMCT.

5 Future Prospects

There are some aspects of the CMCT which can be improved. For example: As Figure 2 shows there are a lot of streets which connect districts at the edge with one district at the center. Reducing the number of streets would lead to a more clear cityspace. This could

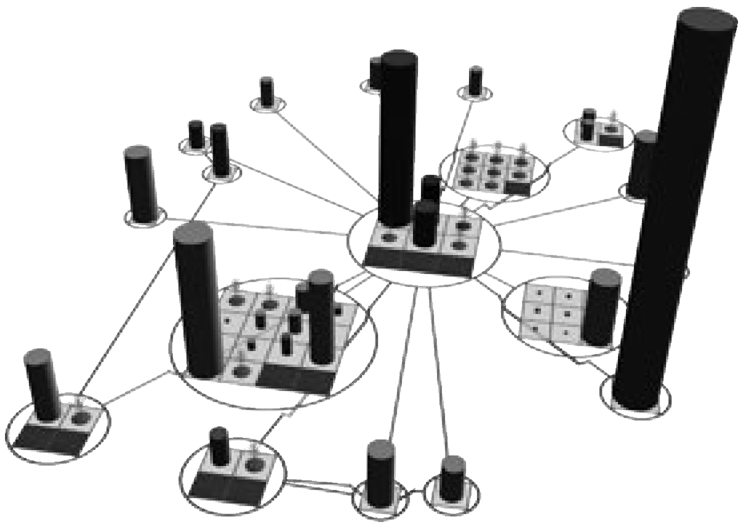


Figure 3: Cityspace of Small Scale Project

be reached with highways. The streets starting in the same area and ending at the same district would form a highway and this highway connects the districts.

A feature for comparing branches of software projects could be implemented for the CMCT. Two or more cities would be displayed side by side (maybe separated by a river) so that the user is able to compare the activities among multiple branches. Code integration from one branch to another could be symbolized with traffic.

6 Conclusion

As this paper shows—and especially Section 4—the conceived concept elements of Section 3 are helpful to comprehend software projects better. It is easy to spot dependencies by the CMCT and it provides a well-arranged cityspace to the user independent of the project size.

During the development of the CMCT we discovered some oddities in the cityspace of the data warehouse project—the cityspace did not match the expected cityspace. By inspecting why the cityspace did not look like expected we discovered that some build files of the project were configured incorrectly. This shows again the power of the CMCT. Developers who have a good knowledge of a software project are able to detect issues they were not aware of.

Finally we summarize the lessons learned from our project: The approach of using the binaries as basic element for visualizing the structure of large scale software projects is

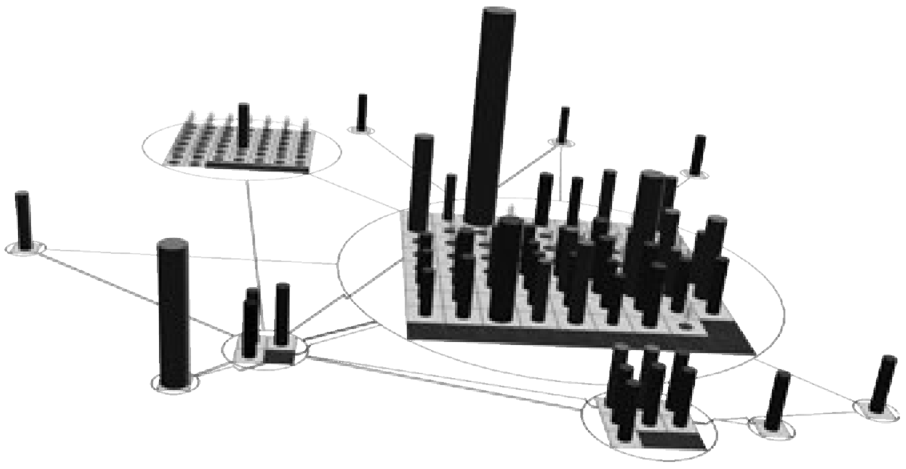


Figure 4: Cityspace of Medium Scale Project

appropriate. Especially for large scale projects this approach reduces the number of visual elements in the city space (see Table 1).

Another advantage of this approach is the independence of a specific technology like in the approaches of [WL07], [SL10], or [KM00] which are limited to Java applications. Inside Microsoft software projects depend on many different technologies (e. g. C, C++, .NET Framework, scripts languages, etc.). For these heterogeneous projects it is hard to find a common element for the visualization (like classes or functions) and if such an element exists for one software project, it is not guaranteed to be the common element in other projects as well.

A major takeaway of the CMCT for software engineers is that the CMCT creates a mental image—[SL10, page 1] shows that a mental image of software is important for the communication among developers. Such images help software engineers to improve their communication inside of software projects. For example, the CMCT was used to explain the structure of the small scale project (see Figure 3) to new developers in the team. Immediately new developers knew the coherence of the single modules and they were able to put their work into the context of the projects architecture.

The CMCT makes refactoring hints available to the engineers as well. As seen in Section 3.2 combinations of different metrics provide those hints (e. g. bad smells). Other combination could point to *AntiPatterns* (see [BMSMM98])—*The Blob* could be identified by a combination of lines of code and number of attributes. It is also possible to extends the collection of metrics with new ones which would forebode to other *AntiPatterns* as well. Addressing other specific software engineering problems with new metrics is possible as well.

Considering the user story "*Clean and Organize Code in Projects*" software engineers are able to evaluate and comprehend the architecture of their projects. How important comprehension of software is showed a Principal Development Lead at Microsoft by approving the value of CMCT: *"There's value in enabling faster code understanding."* In his first job at Microsoft he had to understand 200k lines of code. The CMCT could have helped him to understand all the code in a faster way.

Acknowledgments

We would like to acknowledge to the following persons who have made the completion of this paper possible:

The team (Rich Hanbidge, Olivier Nano) we worked with at the Microsoft Advanced Technology Labs Europe in Aachen, for the support during our research and for making it possible to use the development data of some Microsoft PGs.

Jacek Czerwonka, for providing us with an early development version of the BDB which helped us to develop the CMCT.

References

[AJL$^+$09] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. DebugAdvisor: A Recommender System for Debugging. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 373 – 382. ACM, 2009.

[BKZ10] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125 – 134. ACM, 2010.

[BMSMM98] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1 edition, April 1998.

[BN12] Christian Bird and Nachiappan Nagappan. Who? What? Where? Examining Distributed Development in Two Large Open Source Projects. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 237 – 246, 2012.

[BNDL04] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software Landscapes: Visualizing the Structure of Large Software Systems. In *Proceedings of the Joint Eurographics – IEEE TCVG Symposium on Visualization*, pages 261 – 266, 2004.

[Bro87] Frederick P. Brooks, Jr. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10 – 19, April 1987.

[DMS05] Tim Dwyer, Kim Marriott, and Peter J. Stuckey. Fast Node Overlap Removal. In *Graph Drawing*, pages 153 – 164. Springer, 2005.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[KK89] T. Kamada and S. Kawai. An Algorithm for Drawing General Undirected Graphs. *Inf. Process. Lett.*, 31(1):7 – 15, April 1989.

[KM99] Claire Knight and Malcolm Munroe. Visualizing Software - A Key Research Area. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, Washington, DC, USA, 1999. IEEE Computer Society.

[KM00] Claire Knight and Malcolm Munro. Virtual but Visible Software. In *Proceedings of the International Conference on Information Visualisation*, pages 198 – 205, Washington, DC, USA, 2000. IEEE Computer Society.

[LS09] Claus Lewerentz and Frank Steinbrückner. SoftUrbs: Visualizing Software Systems as Urban Structures. Technical report, BTU Cottbus, February 2009.

[NZZ$^+$10] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change Bursts as Defect Predictors. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, pages 309 – 318, November 2010.

[Pet07] Charles Petzold. *3D Programming For Windows*. Microsoft Press, Redmond, WA, USA, 2007.

[SL10] Frank Steinbrückner and Claus Lewerentz. Representing Development History in Software Cities. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 193–202, 2010.

[SZZ05] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. HATARI: Raising Risk Awareness (Research Demonstration). In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 107 – 110. ACM, September 2005.

[WL07] Richard Wettel and Michele Lanza, editors. *Visualizing Software Systems as Cities*, 2007.

[ZWDZ04] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563 – 572. IEEE Computer Society, May 2004.

Statistical Analysis of Changes for Synthesizing Realistic Test Models

Hamed Shariat Yazdi*, Pit Pietsch*, Timo Kehrer† and Udo Kelter

Software Engineering Group
University of Siegen
Software Engineering Group - University of Siegen
{shariatyazdi, pietsch, kehrer, kelter}@informatik.uni-siegen.de

Abstract: Tools and methods in the context of Model-Driven Engineering have to be evaluated and tested. Unfortunately, adequate test models are scarcely available in many application domains, and available models often lack required properties. Test model generators have been proposed recently to overcome this deficiency. Their basic principle is to synthesize test models by controlled application of edit operations from a given set of edit operation definitions. If test models are created by randomly selecting edit operations, then they become quite unnatural and do not exhibit real-world characteristics; generated sequences of edit operation should rather be similar to realistic model evolution. To this end, we have reverse-engineered a carefully selected set of open-source Java projects to class diagrams and computed the differences between subsequent revisions in terms of various edit operations, including generic low-level graph edit operations and high-level edit operations such as model refactorings. Finally, we statistically analyzed the distribution of the frequency of these edit operations. We have checked the fitness of 60 distributions in order to correctly represent the statistical properties. Only four distributions have been able to adequately describe the observed evolution. The successful distributions are being used to configure our model generator in order to produce more realistic test models.

1 Introduction

Model Driven Engineering has gained a prominent role in the context of software engineering and many tools and methods have been proposed in the last couple of years. These tools and techniques have to be evaluated with regard to aspects such as efficiency, quality of results and scalability. Examples of such tools are model transformations engines, model search engines, model checkers and model versioning systems.

In many application domains real test models are barely available. Real models available often lack characteristics that are required for testing purposes. In fact, the requirements on test models depend a lot on the tool being tested. Some tools, e.g. differencing algorithms, need pairs or sequences of models where the evolution is precisely known while other

*H. Shariat Yazdi and P. Pietsch are supported by the DFG under grant KE 499/5-1.
†T. Kehrer is supported by the DFG under grant KE 499/7-1.

tools just need large test models to assess their efficiency and scalability. As discussed in [PSYK11], artificial models and their evolution should hence meet the following general requirements: **(a)** They should be correct according to their meta model. **(b)** They should satisfy extra constraints, e.g. multiplicities of references. **(c)** In the case of changes, the changes should contain atomic as well as high-level edit operations. **(d)** The changes and evolution should be "realistic" in the sense that they should mimic the statistical properties observable in real models.

The SiDiff Model Generator (SMG) [PSYK12] meets these requirements. It can be configured to create "realistic" test models if the statistical properties of the evolution in real models are observed and known. The SMG modifies a given base input model by applying edit operations, which are based on the meta-model of the input model. The application process is statistically controlled by a component called *Stochastic Controller*. The configuration for the stochastic controller contains random variates of distributions for different edit operations.

State-of-the-art approaches to understand the evolution of models of software systems are based on software metrics and similar static attributes; the extent of the changes between revisions of a software system is expressed as differences of metrics values, and further statistical analyses are based on these differences. Unfortunately, such approaches do not reflect the dynamic nature of changes well. For instance, considering the static metric *Number of Methods* (NOM) of classes: If we observe an increase of one in this metric between two subsequent revisions, the actual amount of change might be much larger, e.g. 5 existing methods deleted, 6 new methods added and 3 methods moved to another class.

This error can be avoided by first computing a precise specification of all changes between two revisions, i.e. a **difference**, and then computing difference metrics [Wen11]. In our above example we would use the difference metrics *NOM-Deleted*, *NOM-Added* and *NOM-Moved* in which we get 14=(5+6+3) changes in total rather than an increase by 1 in the static metric NOM. In other words, we have to count the occurrences of edit operations that have been applied between subsequent revisions of a system.

While this approach seems obvious, it remains to be shown that it can be successfully implemented. The most important question is how differences could be defined. Obviously, textual differences consisting of insertions and deletions of lines of source code will not be a basis for computing meaningful difference metrics. Thus, we reverse-engineered design-level class diagrams from a set of carefully selected open-source Java systems. These class diagrams were compared using advanced model comparison techniques in order to compute changes between revisions on two different levels of abstraction, i.e. based on two different sets of edit operation definitions. The first set of edit operations consists of low-level graph modifications. The second set of edit operations contains high-level edit operations including model refactorings which are applicable to class diagrams from a user's point of view. Details of our approach to modeling and gathering changes of Java software are presented in Section 2. We investigated the fitness of 60 distributions to represent the statistical properties of the observed frequencies, both on low-level edit operations and high-level ones.

The rest of this paper is organized as follows: Section 3 discusses which software repositories could be regarded as candidates for this investigation and justifies our choices. We then introduce the four successful distributions[1] briefly in Section 4. More information for these statistical models can be found on the accompanying website of the paper [SYPKK12]. Section 5 then presents the results of the fitting of the successful distributions to the computed differences. Threads to validity of our work are discussed in Section 6; related work is discussed in Section 7. The paper ends in Section 8 with a summary and a conclusion.

2 Modeling Changes of Java Software

As we are interested in design-level changes, the source code of each revision of a Java software system must first be reverse-engineered into an appropriate model. This is accomplished by a parser that analyzes the structure of the code and creates a class diagram of the Java system. The simplified core of the meta model is depicted in Figure 1, while the complete meta model consists of 15 different element types and is presented in [SYPKK12].

The root element of every model is a project (JProject). Each project can contain a number of packages (JPackage), which in turn can form nested hierarchies. Packages can contain classes (JClass) and interfaces (JInterface). Interfaces can contain only methods (JMethod) and constants (JConstant), whereas classes can additionally contain attributes (JField). Naturally, methods can have parameters (JParameter).

The seven element types that are omitted in Figure 1 represent constructs which are specific to the Java programming language: Primitive types of Java are modeled as simple types (JSimpleType), arrays are represented as special elements (JArrayType). The concept of generics in the Java programming language is modeled by three element types (JGenericType, JTemplateBinding and JTemplateWrapper). Finally, enumerations are represented by two different element types (JEnumeration and JEnumerationLiteral).

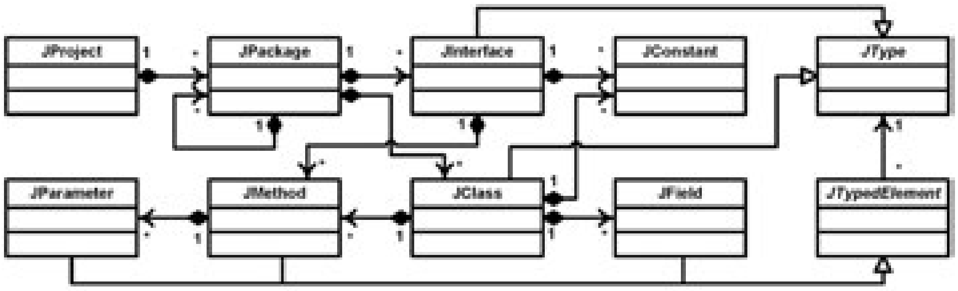


Figure 1: Meta Model for Class Diagrams of Java Source Code - Simplified Core

Having the appropriate class diagrams at hand, a meaningful difference between two revi-

[1]To be precise: Discrete Pareto, Yule, Warring and Beta-Negative Binominal distributions.

sions can be obtained by model comparison technology. A survey on approaches to model comparison can be found in [KRPP09]. Because of the lack of persistent identifiers in reverse-engineered models, we decided to use the SiDiff model differencing framework [KKPS12] in our analysis. We carefully adapted the matching engine to the comparison of the design-level class diagrams. Finally, the changes between revisions are reported on two different levels of abstraction which can be best explained by having a look at the processing pipeline of the SiDiff model differencing framework which is shown in Figure 2.

In the initial matching phase corresponding elements are identified. Based on this matching a low-level difference can then be derived. Generally, five different kinds of low-level edit operations are defined between two subsequent revisions of R_n and R_{n+1}: **Additions:** An element is inserted in R_{n+1}. **Deletions:** An element is removed from R_n. **Moves:** An element is moved to a different position, i.e. the parent element is changed in R_{n+1}. **Attribute Changes:** An element is updated in R_{n+1}, e.g. its name or visibility is changed. **Reference Changes:** A reference of an element is changed, e.g. a method now has a different return type.

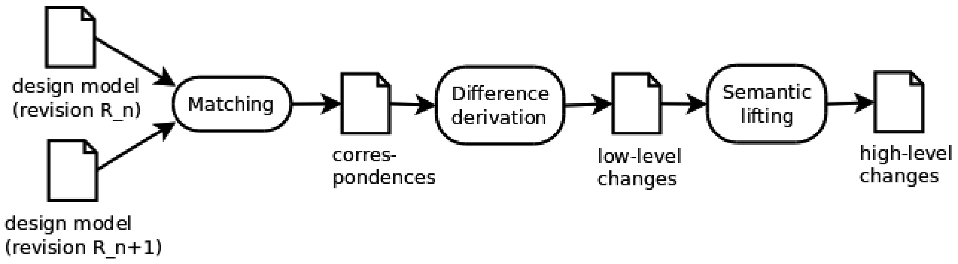


Figure 2: Coarse-grain structure of model comparsion tools

Difference metrics for low-level changes can be computed for each element type and for each kind of edit operations by counting their occurrences in a difference. Thus, we obtain a total number of 75 difference metrics, i.e. 5 difference types times 15 element types.

Low-level changes can be semantically lifted to high-level changes which usually comprise a set of low-level changes. For example, the same result of the low-level changes adding an element of type JField and subsequently setting its JType can be achieved by one high-level operation which takes the JType of the JField to be created as an additional argument and achieves both low-level changes together (see Figure 1). High-level operations such as refactorings can comprise even larger sets of low-level changes. The semantic lifting engine that we have used in our study is presented in [KKT11]. Obviously, the set of high-level operations to be detected has to be defined individually for each modeling language. The operations defined for the class diagrams can be found at the accompanying website [SYPKK12]. In sum, we identified and defined a total number of 188 high-level edit operations in class diagrams, including 12 refactorings. Quantitative measurements of high-level changes can be easily obtained by counting the occurrences of such edit operations.

3 Selection of the Sample Projects

In the previous section, we showed that difference metrics between class diagrams are a new, more fine-grained description of changes in software systems. An investigation of the statistical properties of these difference metrics must be based upon a set of representative sample projects. This section describes and justifies our selection of the sample projects and sketches the technical preparations which were necessary before the statistical analysis could be performed.

We applied the following constraints in the selection of projects: *First*, only real, non-trivial software systems are to be considered. *Secondly*, the projects must be developed over a long period in order to let us study their evolution. *Thirdly*, the selected projects must be typical Java software systems.

We found out that the projects reported in the Helix Software Evolution Data Set (HDS) [VLJ10] fulfill these three requirements. The projects stem from different application domains, have been under development for at least 18 months, have all more than 15 releases and contain at least 100 classes.

We randomly selected nine projects from the HDS, the website of the paper [SYPKK12] provides basic information about the selected projects and their model representations. We checked out all revisions from the respective repositories. After purifying the data from any non-source files, a parser created a class diagram for each revision of each project; totally 6559 models were created. Models of successive revisions were then compared by the SiDiff model differencing framework and the differences between each pair were computed according to the procedure described in Section 2. Finally for each project, the values of the 75 low-level and the 188 high-level difference metrics have been computed serving as input for our statistical analysis. Hence, for each project p and each difference metric m, our data set $\mathcal{S}_{p,m}$ is the calculations of m between all subsequent revisions of p. Additionally, because the models are obtained by reverse engineering the source code of the projects, there are cases that between two subsequent revisions only parts of the system were changed which do not influence the model representation. Therefore if for a given project $\hat{p}$ all of its computed difference metrics are zero between two subsequent revisions, the computed data is excluded from the corresponding data sets $\mathcal{S}_{\hat{p},m}$. We tried to fit each distribution on every $\mathcal{S}_{p,m}$.

4 Statistical Models for Describing Changes

The previous section described how the sample projects were selected and how difference metrics were computed. Our goal was to find statistical models, i.e. distributions, which correctly model the changes observed in our sample data sets. The main challenge for such distributions are large changes: they do happen, but their probabilities are quite small. Suitable distributions must therefore be skewed and asymmetric with heavy tails.

Many continuous and discrete univariate distributions are known [JKB94, JKK05, WA99].

We tested 60 distributions[2]. Only four discrete distributions with heavy tails performed acceptable, although with different levels of success (see Section 5). These four are the discrete Pareto distribution (DPD) of the power law family and the Yule distribution (YD), the Waring distribution (WD) and the Beta-Negative Binomial distributions (BNBD) from the family of hypergeometric distributions.

These successful distributions are briefly introduced here; the inclined reader can find the detailed information of these distributions on the website of the paper [SYPKK12].

Power Law and Discrete Pareto Distribution Considering the function $y = f(x)$, it is said that y obeys **Power Law** to x when y is proportional to $x^{-\alpha}$. Such relations, which have different applications, have been observed in linguistics, biology, geography, economics, physics and also computer science, e.g. the size of computer files, grid complex networks, the Internet and web pages hit rates (see [New05, Mit04, IS10, AH02]).

The discrete Pareto distribution that is used throughout this paper is of power law and is based on the Riemann Zeta function ([EMOT55, GR07, OLBC10]); it is obtained from the General Lerch distribution ([ZA95, WA99, JKK05]). It takes a real value $\rho > 0$ as shape parameter.

Yule, Waring and Beta-Negative Binomial Distributions The Yule distribution, which has applications in taxonomy of species in biology, has just one parameter b which is a positive real. The Waring distribution, which yields the Yule distribution as its special case has two real parameters, $b > 0$ and $n \geq 0$ ([WA99, JKK05]).

Both of the Yule and Waring distributions have been generalized to a hypergeometric distribution called the Beta-Negative Binomial distribution[3] [Irw75]. The distribution has three parameters, α, β and n and is usually denoted by $BNB\left(\alpha, \beta, n\right)$. Its parameters are positive reals [WA99, JKK05]. This distribution can be obtained from the Negative Binomial distribution when the probability p of its Bernoulli trials has the Beta distribution.

5 Analysis of the Data Set and Results

As discussed in Section 3, our data sets ($\mathcal{S}_{p,m}$) were computed considering 263(=75+188) difference metrics for each of the 9 projects through its life span. This section discusses how the four proposed distributions fit to the observed data.

To decide whether or not a distribution $\boldsymbol{\mathcal{D}}$ fits to the data sets $\mathcal{S}_{p,m}$, the null and alternative hypotheses, i.e. $\mathcal{H}_0$ and $\mathcal{H}_1$, are defined as follows: $\boldsymbol{\mathcal{H}_0}$: The data set obeys the distribution $\mathcal{D}$. $\boldsymbol{\mathcal{H}_1}$: The data set does not obey the distribution $\mathcal{D}$.

Different methods exist for fitting distributions and estimating parameters. Two commonly used are *the method of moments* and *the maximum-likelihood estimation method* (MLE). The former tries to estimate the parameters using the observed moments of the sample, by equating them to the population moments and solving the equations for the parameters.

[2]See the accompanying website of the paper at [SYPKK12] for the full list of the tested distributions.

[3]In the literature also referred to as the Generalized Waring distribution and the Beta-Pascal distribution.

The MLE method estimates the parameters by trying to maximize the logarithm of the likelihood function. In this paper the MLE method is employed and the calculations are done using Wolfram Mathematica® 8.0.4 computational engine.

Due to the discrete nature of the difference metrics and the four distributions, the Pearson's Chi-Square test was used. The significance level was set to 0.05. At first the parameters of the desirable distribution $\mathcal{D}$ were estimated, then the p-value of the Pearson's chi-square statistic was calculated in order to decide whether to reject $\mathcal{H}_0$ in favor of $\mathcal{H}_1$ or not.

For the 60 distributions that were initially tested, totally 40500 ($60 \times 9 \times 5 \times 15$)[4] fittings were considered for low-level operations and 101520 ($60 \times 9 \times 188$)[5] for high-level ones. From those, just the results for the four proposed distributions are covered in detail here, separately for low-level and high-level operations.

In the rest of this section only successfully accomplished fittings are reported, i.e. when we were able to decide whether to reject $\mathcal{H}_0$ or not; our summaries of the results are based on such successfully accomplished fittings. There were cases where the computed difference metrics were zero for all revisions and no fittings were possible; they are not considered in our analysis.

Since the number of high-level operations are too many (188), we are unable to fully publish the detailed results. Hence, only the summary of our findings is provided here. The more detailed results are available on the accompanying website of the paper [SYPKK12].

5.1 Fitting the Discrete Pareto Distribution

Since the support of the discrete Pareto distribution consists of positive integers, the fittings are done on the shifted data which are obtained by adding $+1$ to members of our data sets. The shift brings the data in the domain of this distribution.

Low-Level Operations Totally 294 successful fittings were performed for low-level operations for the DPD. $\mathcal{H}_0$ was not rejected 157 times, so the non-rejection ratio is about 53%.

The DPD is most successful in describing changes of packages and interfaces, but with lower success rate for additions of new packages. It has generally a moderate rate of success in describing changes of classes and performs worse when fields are considered. The DPD is not successful in describing changes of methods and parameters due to a success rates of under 30%. Changes of array types could be fully modeled by this distribution. Additions and deletions of other element types could also be described with moderate success. Figure 3 shows one probability plot of the observed and the fitted probabilities for the JFreeChart project for the difference metric additions of methods. The plot is near to the ideal dashed line, so we constitute a good approximation here although $\mathcal{H}_0$ is rejected.

High-Level Operations For the high-level operations $\mathcal{H}_0$ was not rejected 69% of the times; a considerable improvement for the DPD compared to its application on low-level

[4] 60 distributions, 9 projects, 5 low-level operations, 15 model element types.

[5] 60 distributions, 9 projects, 188 high-level operations.

operations. We conclude that the DPD serves better to describe high-level changes.

5.2 Fitting the Yule Distribution

Low-Level Operations From the 294 fittings of the YD, $\mathcal{H}_0$ was rejected in favor of $\mathcal{H}_1$ 180 times, giving a non-rejection rate of almost 39% which makes this distribution the least successful one.

The YD was fully successful in describing moves, reference change and update on packages and interfaces; but for additions and deletions this rate drops to less than 50%. For classes, fields, methods and parameters it performs weakly most of the time, rarely reaching 50% of success. Describing additions of elements is only moderately successful, while deletions of elements are better modeled compared to additions. Nevertheless, the YD performs worse than the DPD for both kinds of edit operations. Figure 4 shows the CDF [6] plot of the observed probabilities and the fitted model, for reference changes of methods in the HSQLDB project. There are large differences between the observed probabilities (blue lines) and those who are obtained by the fitted distribution (red lines), which indicates that the fitting is bad and $\mathcal{H}_0$ is strongly rejected.

High-Level Operations Here, $\mathcal{H}_0$ was not rejected at the rate of 49% for the YD. Although this shows an improvement of 10% compared to low-level operations, the YD also performs the worst for both low-level and high-level operations.

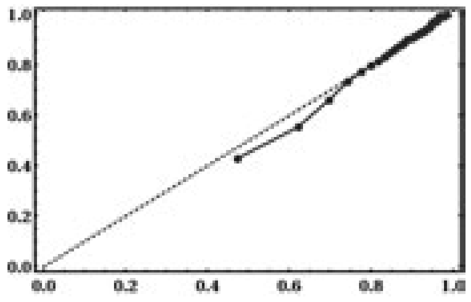


Figure 3: The probability plot of JFreeChart: adding of methods, discrete Pareto distribution.

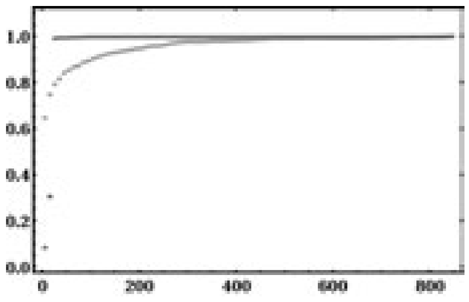


Figure 4: The CDF plot of HSQLDB: reference change of methods, Yule distribution.

5.3 Fitting the Waring Distribution

Low-Level Operations For the WD, $\mathcal{H}_0$ was not rejected 252 out of 294 times, which gives a very good non-rejection rate of 86%.

The WD was fully successful in describing changes of packages and interfaces. For classes

[6]CDF: Cumulative Distribution Function.

it was successful almost 70% of the times or more and this rate is even higher with 90% and more when fields are considered. For changes on methods we get good success rates between 45% to 90%. Figure 5 shows the probability plot for the Maven project considering additions of methods. For changes on parameters this distribution performs also well. The exceptions are reference updates for which it has only a success rate of 25%. It was fully successful in describing changes of array types, constants, simple types, generic types and the other elements (see Section 2).

Figure 7 shows the p-value plot of the DataVision project which shows that the distribution was almost fully successful in describing all kinds of changes on all element types except for reference changes of parameters (row=4, column=6). The black cells emerges from two possibilities: Either there was no data discovered by our change detection tool, i.e. the difference metric value is 0, or the distribution could not be fitted because either the data did not fulfill the requirements of the distribution or parameters could not be estimated. It should be mentioned that the first case happens most of the time, while the second case occurs very rarely. White cells indicate that the calculated p-value was less than the specified significance level, i.e. $\mathcal{H}_0$ was rejected. Finally, when the calculated p-value was above the significance level, i.e. $\mathcal{H}_0$ was not rejected, the cell is colored. The more intense the color of the cell, the higher the p-value.

High-Level Changes The non-rejection ratio of the null hypotheses is at almost 93%, which is obviously a good result. The WD performs very well in describing the change behavior observed in the models almost for all defined high-level operations. This distribution is performing almost 25% better compared to the DPD.

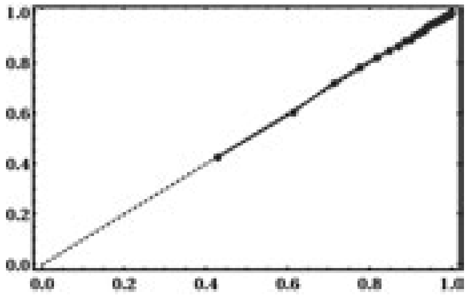


Figure 5: The probability plot of Maven: adding of methods, Waring distribution.

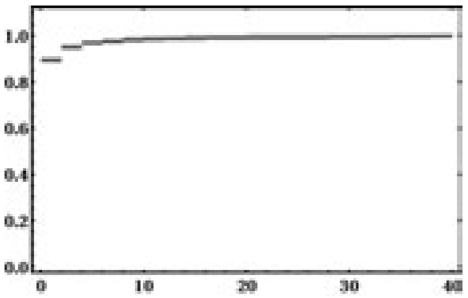


Figure 6: The CDF plot of JFreeMarker: deleting of fields, BNB distribution.

5.4 Fitting the Beta-Negative Binomial Distribution

Low-Level Operations For the BNBD, $\mathcal{H}_0$ was rejected 34 out of 294 times in favor of $\mathcal{H}_1$, yielding an 88% non-rejection rate, which is a slight improvement to the WD.

The performances of the BNBD and the WD are almost identical for the difference metrics over the 15 element types. Like the WD, the BNBD is not successful in modeling reference changes for parameters with a success rate of only about 25%. Figure 6 depicts the CDF plot for the JFreeMarker project considering deletions of fields. It can be seen that the predicted and observed probabilities completely overlap. Figure 8 shows the p-value plot of the Struts project which is almost fully statistically modeled by the BNBD. In this particular example, only reference changes of interfaces could not be modeled (row=4, column=3).

High-Level Operations The non-rejection ratio for the null hypotheses is more than 94% which is quite similar to the WD when high-level operations are considered. Hence, this distribution performs also very well in describing the evolution of class diagrams based on high-level operations.

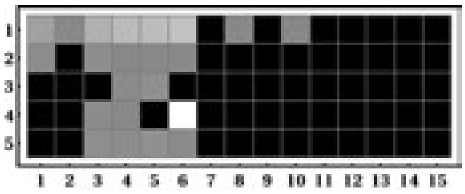


Figure 7: P-Value plot of the whole DataVision project when the Waring distribution is used.

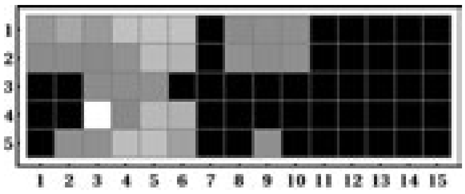


Figure 8: P-Value plot of the whole Struts project when the BNB distribution is used.

Note: In the figures, the rows correspond to change types: 1. Additions, 2. Deletions, 3. Moves, 4. Reference Changes, 5. Attribute Updates. The columns correspond to the element types: 1. Packages, 2. Interfaces, 3. Classes, 4. Fields, 5. Methods, 6. Parameters, 7. Projects, 8. Array Types, 9. Constants, 10. Simple Types, 11. Generic Types, 12. Template Wrappers, 13. Template Bindings, 14. Enumerations and 15. Enumeration Literals.

5.5 Conclusion of Fittings

Considering low-level operations, as discussed in Sections 5.1 and 5.2, the discrete Pareto distribution (DPD) is only to some extend successful in describing the observed changes. The Yule distribution is not generally recommended due to its low success rates.

The DPD is generally good in describing changes on packages and interfaces. It is moderately suitable for classes and fields and only very limited suited for methods and parameters. For the rest of the element types, it performs generally good. When high-level operations are taken into account, the DPD performs much better (near 70%) in describing the changes.

Since the DPD is of power law family, we additionally conclude that the **Power Law** is observable to some extend in low-level changes between class diagrams of open-source Java systems and its presence is more apparent when high-level operations are considered; actually based on the shifted data (see Section 5.1).

Considering low-level operations, the Waring distribution (WD) and the Beta-Negative Binomial distribution (BNBD) show much higher success rates than the other two distributions. Both of them perform equally well at explaining the observed difference metrics for almost all element types. Despite their successes, they are not successful in predicting reference changes of parameters and therefore should be used with caution in this case.

For the high-level operations the success rates of the WD and the BNBD even increases to almost 94%; making them capable of statistically modeling almost any high-level edit operations in addition to low-level ones.

Although the BNBD is an extension to the WD and has one additional shape parameter, this does not add any benefit to its predictive powers. Furthermore, estimating the parameters of the WD needs less effort and is less time consuming.

Comprehensive information about our tests is provided on the website [SYPKK12].

6 Threats to Validity

In this section we discuss threats to the validity of the presented results.

Accuracy One threat is based on the way differences between class diagrams were computed. Model comparison algorithms can produce differences which are generally considered sub-optimal or wrong. [Wen11] has analyzed this error for class diagrams and the SiDiff differencing framework [KKPS12]; the total number of errors was typically below 2%. This very low error rate cannot have a significant effect on the results of our analysis. The second threat is how accurate the high-level operations are recognized. If model elements were matched based on persistent identifiers, the operation detection could be guaranteed to deliver correct results [KKT11], i.e. all low-level changes were grouped into high-level operations and no low-level ones remain ungrouped. As matchings are computed based on similarity heuristics, possible "incorrect" matches can lead to false negatives, i.e. edit operations which have actually been applied but which were not detected. We calculated the rate of ungrouped low-level changes which was below 0.3%, thus both of the difference derivation and semantic lifting engines in our pipeline (see Figure 2) performed quite well and the results are not distorted.

External Validity Another important question is whether our results are generalizable. Our test data set consists of medium-sized, open-source Java software systems. It is highly probable that our results also hold for large Java software systems as our preliminary studies show. It is not yet clear whether our results also hold for closed software systems, in particular if company-specific programming styles and design rules are enforced.

It is also less than clear whether our results hold for other object-oriented languages, e.g. C++. The question is whether the data model for class diagrams (see Figure 1) is still appropriate. These questions are subject of further research.

7 Related Work

Vasa et al. [VSN07] studied the evolution of classes and interfaces in open-source Java software based on static metrics for released versions of the systems. They showed that the average metric values are almost stable over the histories, i.e. the average size and popularity of classes does not change much between released versions. They also analyzed which kind of classes tend to change more. For this, correspondences between classes are established based on their fully qualified name. The amount of change is measured based on value changes of 25 static software metrics. They showed that more complex classes are more likely to undergo changes. This research is continued in [VSNW07], where they consider additional static metrics. Here they show that the majority of changes happen on a small portion of classes. They also analyze the history of classes superficially based on a comparison between the static metric values counted in the final version of the system and those counted in preceding versions. In [Vas10] Vasa presented an extended and more detailed version of his research.

All this research mentioned above, is based only on changes of static metric values and does not take the evolution, i.e. the actual changes between two versions, into account. Furthermore, only released versions of the software systems are considered, i.e. the time period between two versions is rather long. In contrast we used much finer time intervals, i.e. revisions, which reflect the changes more accurately. Lastly, no parametric distribution is reported in any of the publications.

All following papers focus on finding occurrences of distributions for static metric values on software systems. They also have in common that they only use single system snapshots as the base for their analysis, i.e. the topic of system evolution is not brought up at all. Additionally, neither of them tried nor proposed the Yule, Waring and Beta-Negative Binomial distributions in their researches.

Concas et al. [CMPS07] studied metrics of VisualWorks Smalltalk and compared them to those observed in Eclipse and the JDK. They showed that the power law and the continuous univariate Pareto distribution are observable in their data.

Baxter et al. [BFN$^+$06] studied the structure of Java software systems based on static software metrics. They report that some, but not all, of the considered metrics follow power law.

Wheeldon and Counsell [WC03] analyzed power law distributions for different forms of object-oriented couplings in Java software. They extracted graph structures from the source code representing the couplings, e.g. inheritance or return type usage and counted static metrics on them. They identified 12 coupling-related metrics that obey power law.

8 Summary and Conclusion

In this paper we thoroughly studied the evolution of reverse engineered class diagrams based on difference metrics of low-level and high-level edit operations.

Nine typical open-source Java projects were initially selected and a parser created design-level class diagram representations of 6559 source code revisions. All subsequent class diagrams were compared by a model differencing framework. On each calculated difference, 75 low-level and 188 high-level difference metrics were counted.

We then addressed the question which statistical model would be the best to describe the observed changes. Sixty continuous and discrete univariate distributions have been tested and only four of them performed acceptable. These are the discrete Pareto, Yule, Waring and Beta-Negative Binomial distributions. The Yule distribution is generally not recommended due to low success rates. The discrete Pareto distribution showed an acceptable performance on low-level changes and describes changes on high-level changes quite well at almost 70% success rate. Additionally we conclude the presence of the **Power Law** to some extend in the analyzed difference metric values when shifted (see Section 5.1).

The Waring and the Beta-Negative Binomial distributions are the most successful distributions. They can describe almost any type of low-level change for each element type with an success rate near to 90%. The only exceptions are reference changes for parameters. For high-level changes these two distributions perform even better reaching success rates of 93% and 94% respectively. They are capable of modeling almost any high-level changes.

The knowledge of this research is directly used in the SiDiff Model Generator [PSYK12] to create synthetic models emulating realistic evolution of software systems.

References

[AH02] Lada A. Adamic and Bernardo A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3:143–150, 2002.

[BFN$^+$06] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. *SIGPLAN Not.*, 41, 2006.

[CMPS07] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-Laws in a Large Object-Oriented Software System. *IEEE Tran. Software Engineering*, 33, 2007.

[EMOT55] Arthur Erdelyi, Wilhelm Magnus, Fritz Oberhettinger, and Francesco G. Tricomi. *Higher transcendental functions Vol.1*, volume 1. McGraw-Hill, 1955.

[GR07] I. S. Gradshteyn and I. M. Ryzhik. *Table of Integrals, Series and Products, 7th Edition*. Academic Press, 2007.

[IMI08] M. Ichii, M. Matsushita, and K. Inoue. An exploration of power-law in use-relation of Java software systems. In *19th Australian Conf. Software Engineering ASWEC*, 2008.

[Irw75] Joseph Oscar Irwin. The Generalized Waring Distribution. Part I. *Journal of the Royal Statistical Society. Series A (General)*, 138(1):pp. 18–31, 1975.

[IS10] Lovro Ilijašic and Lorenza Saitta. Long-tailed distributions in grid complex network. In *Proc. 2nd Workshop Grids Meets Autonomic Computing GMAC*, USA, 2010. ACM.

[JKB94] Norman L. Johnson, Samuel Kotz, and N. Balakrishnan. *Continuous Univariate Distributions, Volume 1 & Volume 2*. Wiley, 2nd edition, 1994.

[JKK05] Norman L. Johnson, Samuel Kotz, and Adrienne W. Kemp. *Univariate Discrete Distributions*. Wiley Interscience, 3rd editon edition, 2005.

[KKPS12] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt. Adaptability of model comparison tools. In *Proc. 27th Inter. Conf. Automated Software Engineering ASE*, USA, 2012.

[KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Proc. 26th IEEE/ACM Inter. Conf. Automated Software Engineering ASE*, USA, 2011.

[KRPP09] D. S. Kolovos, D. Di Ruscio, R. F. Paige, and A. Pierantonio. Different models for model matching: An analysis of approaches to support model differencing. In *Proc. ICSE Workshop Comparison & Versioning of Software Models CVSM*, USA, 2009.

[Mit04] Michael Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1:226–251, 2004.

[New05] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, December 2005.

[OLBC10] Frank W. J. Olver, Daniel W. Lozier, Ronald F. Boisvert, and Charles W. Clark. *NIST Handbook of Mathematical Functions*. NIST and Cambridge Uni. Press, 2010.

[PSYK11] Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. Generating realistic test models for model processing tools. In *26th Inter. Conf. Automated Software Engineering ASE*, USA, 2011.

[PSYK12] Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. Controlled Generation of Models with Defined Properties. In *Software Engineering SE2012*, Berlin, Germany, 2012.

[SYPKK12] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. Accompanied material and data for the SE2013 paper. `http://pi.informatik.uni-siegen.de/qudimo/smg/se2013`, 2012.

[Vas10] Rajesh Vasa. *Growth and Change Dynamics in Open Source Software Systems*. PhD thesis, Swinburne University of Technology, 2010.

[VLJ10] Rajesh Vasa, Markus Lumpe, and Allan Jones. Helix - Software Evolution Data Set. `http://www.ict.swin.edu.au/research/projects/helix`, 2010.

[VSN07] Rajesh Vasa, Jean-Guy Schneider, and Oscar Nierstrasz. The Inevitable Stability of Software Change. In *IEEE Inter. Conf. Software Maintenance, ICSM*, 2007.

[VSNW07] Rajesh Vasa, Jean-Guy Schneider, Oscar Nierstrasz, and Clinton Woodward. On the Resilience of Classes to Change. *ECEASST*, 8, 2007.

[WA99] Gejza Wimmer and Gabriel Altmann. *Thesaurus of Univariate Discrete Probability Distributions*. Stamm, 1st edition, 1999.

[WC03] Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. In *Proc. 3rd IEEE Inter. Workshop Source Code Analysis & Manipulation*. IEEE, 2003.

[Wen11] Sven Wenzel. *Unique Identification of Elements in Evolving Models: Towards Fine-Grained Traceability in Model-Driven Engineering*. PhD thesis, Uni. Siegen, 2011.

[ZA95] Peter Zörnig and Gabriel Altmann. Unified representation of Zipf distributions. *Computational Statistics and Data Analysis*, 19(4):461 – 473, 1995.

Experience from Measuring Program Comprehension—Toward a General Framework

Janet Siegmund*$^{\sigma}$, Christian Kästner$^{\omega}$, Sven Apel$^{\pi}$, André Brechmann$^{\theta}$, Gunter Saake$^{\sigma}$
$^{\sigma}$University of Magdeburg, $^{\omega}$Carnegie Mellon University, Pittsburgh
$^{\pi}$University of Passau, $^{\theta}$Leibniz Institute for Neurobiology, Magdeburg

Abstract: Program comprehension plays a crucial role during the software-development life cycle: Maintenance programmers spend most of their time with comprehending source code, and maintenance is the main cost factor in software development. Thus, if we can improve program comprehension, we can save considerable amount of time and cost. To improve program comprehension, we have to measure it first. However, program comprehension is a complex, internal cognitive process that we cannot observe directly. Typically, we need to conduct controlled experiments to soundly measure program comprehension. However, empirical research is applied only reluctantly in software engineering. To close this gap, we set out to support researchers in planning and conducting experiments regarding program comprehension. We report our experience with experiments that we conducted and present the resulting framework to support researchers in planning and conducting experiments. Additionally, we discuss the role of teaching for the empirical researchers of tomorrow.

1 Introduction

Today, we are surrounded by computers. They are in our cars, credit cards, and cell phones. Thus, there is a lot of source code that needs to be implemented and maintained. In the software-development life cycle, maintenance is the main cost factor [Boe81]. Furthermore, maintenance developers spend most of their time with understanding source code [vMVH97, Sta84, Tia11]. Thus, if we support program comprehension, we can save considerable amount of time and cost of the software-development life cycle.

To improve program comprehension, various programming techniques and tools were improved since the first programmable computers. From machine code over assembly languages, procedural programming, and contemporary object-oriented programming [Mey97], modern programming paradigms, such as feature-oriented and aspect-oriented programming emerged [Pre97, KLM^{+}97]. In the same way, tools have been developed to support programmers in working with source code. For example, Eclipse, FeatureIDE [KTS^{+}09], or CIDE [KAK08] target, among others, better comprehension of software.

These new techniques and tools are only rarely evaluated empirically regarding their

*This author published previous work as Janet Feigenspan effect on program comprehension. However, program comprehension is an internal cognitive process and can be evaluated only in controlled experiments—plausibility arguments are not sufficient, because they can prove wrong in practice. One reason for the reluctance of using empirical evaluation is the effort for conducting experiments, which usually takes several months from the planning phase to results.

With our work, we want to raise awareness of the necessity of empirical research and initiate a discussion on how to motivate and support other researchers to conduct more empirical studies in software engineering. We focus on the following contributions:

- Description of typical problems when planning controlled experiments based on our experience.
- Framework to reduce the effort of conducting controlled experiments.
- Establish and revive the empirical-research community in software engineering.

Of course, the appeal for conducting more empirical research in software engineering is not new [Kit93, TLPH95, Tic98]. Nevertheless, in a recent study, Sjoberg and others found that the amount of papers reporting controlled experiments is still low (1.9 %). Thus, there is still a reluctance to apply empirical methods. By reporting our experience and the resulting framework, we hope to remove the obstacles of planning and conducting experiments and, thus, motivate other researchers to apply controlled experiments to evaluate their new techniques and tools regarding the effect on programmers.

This paper is structured as follows: First, we take a closer look at program comprehension and the logic of experiments, followed by our experience with program-comprehension experiments. This way, we want to raise the awareness for the difficulties that accompany controlled experiments. Then, we present the framework, which we developed based on our experience and which supports researchers with conducting experiments. All material we present here is also available at the project's website (`http://fosd.net/experiments`).

2 Program Comprehension and its Measurement

In this section, we recapitulate important concepts related to program comprehension to ensure the same level of familiarity for our readers. Readers familiar with program comprehension and controlled experiments may skip this section. Our intention is not to give an exhausting overview of program comprehension, but to give an impression of the complexity of the process. We are aware that we are focusing on one aspect of program comprehension (other aspects are described, e.g., by Fritz and others [FOMMH10]). In the same way, we only shortly discuss software measures.

To understand source code, developers typically use either top-down or bottom-up comprehension. When developers are familiar with a program's domain, they use top-down comprehension; so they start with stating a general hypothesis about a program's purpose [Bro78, SV95, SE84]. By looking at details, developers refine this hypothesis. If they encounter a part of a program from an unfamiliar domain, they switch to bottom-up comprehension. In this case, developers analyze source code statement by statement and group statements to semantic chunks, until they understand the code fragments [Pen87, SM79].

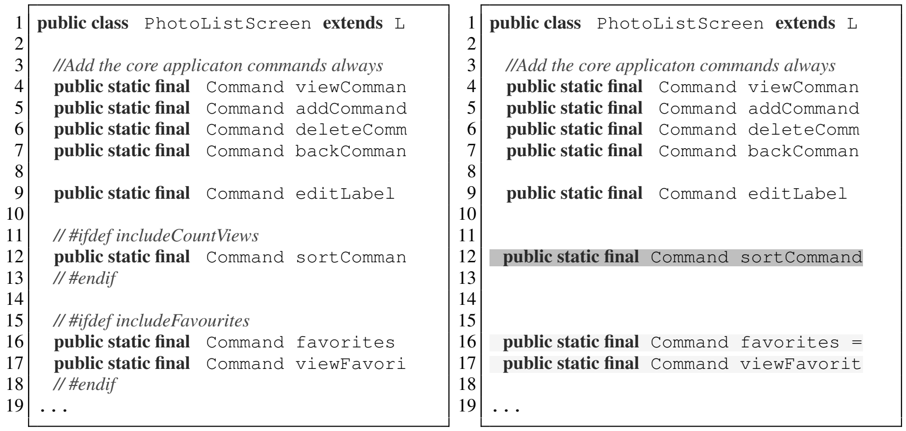
Thus, program comprehension is a complex process. To measure it, we need to find an indicator. Often, software measures are used, such as lines of code or cyclomatic complexity [HS95]. It seems reasonable that the more lines of code or the more branching statements a program has, the more difficult it is to understand. However, software measures cannot fully capture the complex process of understanding source code [Boy77, FALK11]. Thus, we should not rely solely on software measures to measure program comprehension.

Another way is to observe in controlled experiments how human participants understand source code. This way, we do not rely on properties of source code, but consider the comprehension process itself. Often, tasks, such as corrective or enhancing maintenance tasks are used [DR00, FSF11]. The idea is that if participants succeed in solving a task, they must have understood the source code—otherwise, they would not be able to provide a correct solution. Another way is to use think-aloud protocols, in which participants verbalize their thoughts [ABPC91]. This allows us to observe the process of comprehension. Both techniques, tasks and think-aloud protocols only indirectly assess program comprehension. So far, there is no way to look into participants' brain while they are understanding source code.

Designing controlled experiments with human participants requires considerable effort and experience, because there are *confounding* parameters that need to be considered. Confounding parameters *influence* the behavior of participants and can bias the result. For example, an expert programmer understands source code different than a novice programmer; a participant who is familiar with a program's domain uses top-down comprehension, whereas a participant who is unfamiliar with a domain uses bottom-up comprehension. Even details that appear irrelevant may influence behavior, such as violating coding conventions (the performance of expert programmers can decrease to the level of novice programmers [SE84]) or using under_score vs. camelCase identifier style [BDLM09, SM10].

After identifying relevant confounding parameters, we need to select suitable control techniques. For example, to avoid bias due to differences in programming experience, we could only recruit novice programmers. However, this would limit the applicability of our results to novice programmers. Instead of recruiting only novices, we could also measure programming experience and evaluate how it influences the result. However, measuring programming experience takes additional time and requires that a measurement instrument exists. Additionally, we need more participants and it increases the complexity of the experimental design and analysis methods. Thus, there is always a trade off between generalizability, accuracy of results, and available resources.

Next, we present our experiences with our controlled experiments and how we addressed this trade off.



```
1  public class PhotoListScreen extends L
2
3    //Add the core applicaton commands always
4    public static final Command viewComman
5    public static final Command addCommand
6    public static final Command deleteComm
7    public static final Command backComman
8
9    public static final Command editLabel
10
11   // #ifdef includeCountViews
12   public static final Command sortComman
13   // #endif
14
15   // #ifdef includeFavourites
16   public static final Command favorites
17   public static final Command viewFavori
18   // #endif
19 ...
```

```
1  public class PhotoListScreen extends L
2
3    //Add the core applicaton commands always
4    public static final Command viewComman
5    public static final Command addCommand
6    public static final Command deleteComm
7    public static final Command backComman
8
9    public static final Command editLabel
10
11
12  public static final Command sortCommand
13
14
15
16  public static final Command favorites =
17  public static final Command viewFavorit
18
19 ...
```

Figure 1: Left: Source code with #ifdef directives; right: source code with background colors. In the colored version, Line 12 is annotated with orange background color, Lines 16 and 17 with yellow.

3 Experience

When we started our work on program comprehension in 2009, we set out to evaluate how modern programming paradigms, such as feature-oriented programming [Pre97] or aspect-oriented programming [KLM$^+$97], and new tools, such as CIDE [KAK08], influence program comprehension. However, during the planning phase, we soon learned that such broad questions are unsolvable in a single experiment—we would need over one million participants to account for all confounding parameters [Fei09].

Thus, we narrowed our research question. In a first experiment, we evaluated how background colors support program comprehension compared to no background colors in preprocessor-based code [FKA$^+$12]. As material, we used one medium-sized Java program that was implemented with preprocessor directives.[1] From that program, we created a second version, in which we used background colors instead of preprocessor directives. Everything else was the same. In Figure 1, we show a screen shot of both version to give an impression.

By narrowing our research questions, we controlled for a lot of confounding parameters and can attribute program comprehension, operationalized by correctness and response time of solutions, only to the difference in the source code we used (i.e., background colors vs. textual #ifdef directives). As a drawback, however, our results are limited to the circumstances of our study: Java as the programming language, medium-sized program, students as participants, etc.

Designing this seemingly simple study took us several months and a master's thesis. The

[1]Java is a popular language to develop Apps for mobile devices. To meet the resource constraints, preprocessors, such as Antenna or Munge, are also used for Java.

```
public class PhotoListScreen{
  /* ... */
  // #ifdef includeFavourites
  public static final Command favoriteCommand = new
          Command("Set Favorite",Command.ITEM,1);
  public static final Command viewFavoritesCommand = new
          Command("View Favorites",Command.ITEM,1);
  // #endif
  /* ... */
  public void initMenu() {
    /* ... */
    // #ifdef includeFavourites
    this .addCommand(favoriteCommand);

    // #endif
    /* ... */
  }
}
```

Figure 2: Bug for M3: `viewFavoritesCommand` is not added.

most difficult problems were to find suitable material, present the material to participants, control for programming experience as one of the most important parameters, and find a suitable indicator for program comprehension. We discuss each problem in more detail.

First, we needed to find suitable material. Participants should not understand it at first sight nor be overwhelmed by the amount of source code. Furthermore, the source code should be implemented in Java, because participants are familiar with it. After several weeks of searching and comparing source code, we found MobileMedia, which has a suitable size, was code reviewed to ensure coding conventions, and was implemented in Java ME with preprocessors [FCM$^+$08]. Hence, finding suitable material is tedious and can take some time.

Second, we needed to present source code to participants. If we had used Eclipse, participants who were familiar with it (e.g., knew how to use call hierarchies or regular expressions to search for code fragments), would have performed better independent of whether they worked with background colors of #ifdef directives. Eventually, we used a browser setting with syntax highlighting and basic source-code navigation, but no other tool support. Thus, an intuitive solution may not be optimal and a seemingly awkward solution may be the better choice.

Third, we needed to control for programming experience, because participants with more experience typically understand source code better. Since we did not find any validated questionnaire or test to measure programming experience, we constructed our own based on a literature survey and by consulting programming experts. We again needed to investigate a couple of weeks, but as result we had a questionnaire that we can reuse in subsequent experiments. Hence, we should take great care to control for confounding parameters, especially if we believe they have an important influence on our result.

Last, to measure program comprehension, we used tasks, which participants could only solve if they understood the underlying source code first. In a pilot study, we observed that our tasks fulfilled that criterion, but some of them were too difficult. To give an impression of the nature of the tasks, we present the source code of one task in Figure 2. The bug description was that a command was not shown, although it is implemented. The bug was located in Line 12, where the command was not added to the menu. Thus, we adapted the tasks and evaluated their difficulty in a second pilot study. Eventually, after several months of careful planning, the design of our study was finished.

In subsequent experiments [FSP+11, FALK11, FKL+12, SFF+11, FKA+12, SBA+12, SKLA12], we did profit from our experiences of the first experiment. For example, we often used different variants of MobileMedia and the programming-experience questionnaire. Furthermore, we developed a feeling for how difficult tasks can be when working with students of computer science, so that often one small pilot study sufficed. Additionally, we could reuse the scripts for analyzing the data. Thus, the effort and time invested in the first experiment payed off for subsequent experiments. With our work, we aim at reducing the effort of the first experiment for other researchers, for whom we developed the framework that we present next.

4 A Framework to Support Controlled Experiments

Based on our experiences, we developed a framework to support researchers to plan and conduct experiments in the context of program comprehension. It consists of four parts: First, we developed a questionnaire to reliably measure programming experience. Second, we implemented a tool for presenting source code, tasks, and questionnaires to participants. Third, we documented confounding parameters for program comprehension. Last, we developed teaching material and holding according lectures to train the empirical researchers of tomorrow. We discuss each part of the framework in more detail in this section.

4.1 Programming-Experience Questionnaire

The first part of our framework is a questionnaire to measure programming experience, one of the major confounding parameters for program comprehension. To this end, we conducted a literature survey of seven major journals and conferences of the last ten years [FKL+12]. We reviewed all papers that reported program-comprehension experiments and extracted how programming experience was defined and measured. Based on these insights, we refined the questionnaire we developed for our first experiment.

It consists of the following four categories (in Appendix 8.1, we show the complete questionnaire):

- Years (related to the amount of time participants spent with programming)
- Education (related to experience participants gained from education)
- Self estimation (participants had to estimate their experience with several topics)

No.	Question	ρ	N
Self estimation			
1	s.PE	.539	70
2	s.Experts	.292	126
3	s.ClassMates	.403	127
4	s.Java	.277	124
5	s.C	.057	127
6	s.Hasekll	.252	128
7	s.Prolog	.186	128
8	s.NumLanguages	.182	118
9	s.Functional	.238	127
10	s.Imperative	.244	128
11	s.Logical	.128	126
12	s.ObjectOriented	.354	127
Years			
13	y.Prog	.359	123
14	y.ProgProf	.004	127
Education			
15	e.Years	-.058	126
16	e.Courses	.135	123
Size			
17	z.Size	-.108	128

ρ: Spearman correlation; N: number of subjects;
gray cells denote significant correlations ($p < .05$).

Table 1: Spearman correlations of number of correct answers with answers in questionnaire.

- Size (size of projects participants had worked with)

To evaluate whether this questionnaire is suitable for measuring programming experience, we evaluated it in a controlled experiment with over 100 undergraduate computer-science students. Specifically, we compared the answers of students in the questionnaire with the number of correct answers for ten programming tasks—the more tasks participants are able to solve correctly within the given time frame (40 minutes), the higher their programming experience should be, and the higher they should estimate their experience in the questionnaire. In Table 1, we show the correlations with the number of correct answers with each question in the questionnaire (in Table 4 in the Appendix, we explain the abbreviations.). A high correlation indicates that a question is suitable to describe programming experience (operationalized by the number of correct answers).

So far, self estimation appears to be a good indicator to assess programming experience. However, questions also correlate among each other. For example, participants who estimate a high experience with logical programming also estimate a high experience with Prolog (a logical programming language typically taught at German universities). Thus, using simply all questions with a high correlation with the number of correct answers is not sufficient. Instead, we need to use *partial correlations*, which is the correlation of two variables that is *cleaned* by the influence of a third variable [Bor04].

Thus, to extract the most relevant questions that best describe programming experience, we used stepwise regression, which is based on partial correlations of variables. With stepwise regression, we identified two relevant questions: The self-estimated experience with logical programming and the self-estimated experience compared to the class mates of students. These two questions can be supplied with control questions (e.g., self estimated experience with Prolog or self estimated programming experience) and used to measure the programming experience of participants. In future work, we plan to further validate our questionnaire and confirm that these two questions are the best indicator for programming experience.

4.2 Program-Comprehension Experiment Tool

As a second part of our framework, we developed the tool PROPHET to present source code, questionnaires, and tasks to participants [FS12]. It is a complex tool for planning and conducting experiments. To support a variety of program-comprehension experiments, we consulted the papers of our literature review again and analyzed the requirements for conducted experiments. Based on these requirements, we implemented PROPHET, which is available at the project's website.

PROPHET has two views, one for the experimenter and one for the participants. In the experimenter view, experimenters can decide how to present source code to participants by selecting check boxes (e.g., with or without syntax highlighting or allowing a search function or not). To give an impression, we show one tab of the experimenter view in Figure 3. In Appendix 8.2, we show additional screen shots.

To customize how participants see source code and tasks, experimenters select check boxes in the preferences tab of the experimenter view, shown in Figure 3 of the Appendix. For example, experimenters can define a file that is displayed when a task begins, choose what behavior of participants to log, whether participants see line numbers or are allowed to use the search, as well as specify a time limit for a task or the complete experiment.

In the view for participants, source code is presented as specified by the experimenter. In a second window, the tasks, questions, and forms for answers are presented.

We used PROPHET for our experiments since 2011 and found it very helpful to prepare the experiments. We did not have to implement any new source code or adapt source code of our existing tool infra structures. Instead of days to prepare the tasks and questions, we now need only hours. Thus, PROPHET saved us a considerable amount of time. We also found that other researchers used PROPHET for their experiments, indicating that PROPHET is general enough to support other researchers. We encourage researchers to use PROPHET for their experiments and give us feedback about their experience.

4.3 List of Confounding Parameters

The next part of our framework is a list of confounding parameters for program comprehension. To this end, we again consulted the paper of our literature review and, this time, extracted parameters that authors treated as confounding parameters.

We identified two categories of parameters: personal and experimental parameters. First, personal parameters are related to the participants, such as programming experience, intelligence, or domain knowledge. In total, we extracted 16 personal parameters. Second, experimental parameters are related to the setting of the experiment, such as the familiarity of participants with tools used, fatigue, or the layout of the study object. We found 23 experimental parameters.

In Appendix 8.3, we present an overview of all currently identified confounding parameters. To give an impression of the nature of confounding parameters, we present the most important parameters based on how often researchers considered them in their study. First, in 112 (of 158) papers, programming experience was mentioned as confounding parameter. Programming experience describes the experience participants have with implementing and understanding source code. The higher the experience, the better participants comprehend source code. Second, familiarity of study object (76) describes how familiar participants are with the concepts being studied, such as Oracle or UML. The more familiar they are, the better they might perform in an experiment. Both, programming experience and familiarity with study object, are personal parameters.

The third parameter is the programming language (72). Participants who are familiar with a language perform different than participants who are not. Fourth, the size of the study object (67) describes how large the study object is, for example in terms of lines of code or number of classes. Last, learning effects were mentioned in 65 papers, which describe that participants learn during an experiment. Thus, they might perform better at the end of an experiment, because they learned how to solve tasks.

With a list in which we describe each possible parameter, researchers can decide for each parameter on the list whether it is relevant for their study and select a suitable control technique. They do not have to put too much effort in identifying the parameters. When we plan our experiments, we are now traversing the list and discuss for each parameter whether it is relevant or not. This saved us considerable time and effort.

To describe how confounding parameters are managed, we suggest a pattern like the one in Table 2. It contains the applied control technique and rationale for the decision, as well as the used measurement technique and the rationale for the technique. This way, readers of a report can get a quick overview of how confounding parameters were managed.

This work will be continued as long as researchers publish experiments, because there might always be parameters that have not been considered so far. Thus, the list of confounding parameters is intended to grow. On the project's website, we present the status of the work, including the currently reviewed papers and extracted parameters. We explicitly encourage other researchers to extend the review with new papers of new venues.

Parameter (Abbreviation)	Control technique		Measured/Ensured	
	How?	Why?	How?	Why?
Programming experience (PE)	Matching	Major confound	Education level	undergraduates have less experience than graduates
Rosenthal effect (RE)	Avoided	Reliable	Standardized instructions	Most reliable
Ties to persistent memory (Ties)	Ignored	Not relevant	—	—

Table 2: *Pattern to describe confounding parameters.*

4.4 Teaching Material

In the German computer-science curricula, empirical methods are under-represented or even neglected at most universities. However, empirical methods are an important aspect also beyond computer science. For example, in psychology, students learn from the first semester how to plan experiments, how to solve the difficulties, how to analyze data, how to develop questionnaires, and so forth.

To improve the current situation, we designed a lecture, in which we teach students the basic methods of empirical research. For example, we teach how to apply the think-aloud protocol or how to set up performance measurements. Furthermore, we teach methods to analyze the data and conduct hypothesis tests to differentiate a random effect from a real effect. So far, students were interested and engaged in the topics of the lecture.

To let students apply the learned methods, students designed, conducted, and analyzed their own experiments and submitted a report. When looking at the report of students, we found that the experiments were carefully designed and analyzed and that some of the experiments are even good enough to be published. Thus, there is evidence that there is a need and interest to learn and teach empirical methods. In the future, we hope to motivate students to select an empirical topic for their bachelor or master's thesis and maybe continue to use empirical methods in a PhD thesis.

So far, this lecture took place at the Philipps University Marburg (held by Christian Kästner). Currently, the lecture takes place at the University of Magdeburg (held by Janet Siegmund). Since we had positive feedback from our students and since we could reuse the material of the lecture, we are planning to offer the lecture again. So far, the lecture is planned at the University of Passau and Carnegie Mellon University. We also are happy to share our experience and teaching material with other researchers to support others in training the empirical researchers of tomorrow.

5 Applying the Framework

To show that our framework helps to conduct experiments, we discuss how it helped us to design two of our experiments. First, we set up and pilot tested an experiment to evaluate how physical and virtual separation of concerns affect program comprehension [SKLA12]. We could consult the list of confounding parameters and decide for each parameter how important its influence is and how we can control for it. We also could apply the questionnaire to measure programming experience and create balanced groups with a comparable experience level. Third, for presenting the task material, and programming-experience questionnaire, we could use PROPHET. The pilot study was conducted at the University of Passau, without the responsible experimenter being present. Instead, a colleague in Passau conducted the experiment without any difficulties.

Second, we conducted an experiment to evaluate whether the derivation of a model is easier to understand than the model itself [FBR12]. We could also consult the list of confounding parameters, because the selection criteria for papers of the literature survey were broad enough to also include experiments on model comprehension. We could also reuse the questionnaire for programming experience, but had to add questions related to model comprehension. We could not use the tool PROPHET, because it does not support modifying images, which was one of the tasks—currently, we can only *display* images. However, we are working on PROPHET to also support the modification of images.

In summary, we now only needed weeks instead of months to set up the experiments. We also encourage other researchers to use the framework and give us feedback how it helped

6 Conclusion and Vision

Although there is effort to establish an empirical research community, empirical research is still reluctantly applied to evaluate new techniques and tools that target, among others, better comprehensibility. We believe that one reason is the effort that accompanies applying empirical methods, such as controlled experiments.

Based on our experience, we developed a framework to help other researchers in overcoming the obstacle of designing the first controlled experiment. The framework consists of a list of confounding parameters and a questionnaire to measure programming experience, the most important confounding parameter. Furthermore, we developed the tool PROPHET to support researchers during planning and conducting experiments. Additionally, we invest our effort in training the empirical researchers of tomorrow.

In future work, we want to explore further strategies to measure program comprehension. In cognitive neuro science, researchers successfully use *functional magnetic resonance imaging* to visualize cognitive processes. In collaboration with André Brechmann, a neuro biologist, we are currently exploring whether functional magnetic resonance imaging can also be used to measure program comprehension [SBA+12].

7 Acknowledgments

Thanks to Norbert Siegmund for fruitful discussions. Siegmund's and Saake's work is funded by BMBF project 01IM10002B, Kästner's work partly by ERC grant #203099, and Apel's work by the German Research Foundation (DFG – AP 206/2, AP 206/4, and LE 912/13).

References

[ABPC91] Neil Anderson, Lyle Bachman, Kyle Perkins, and Andrew Cohen. An Exploratory Study into the Construct Validity of a Reading Comprehension Test: Triangulation of Data Sources. *Language Testing*, 8(1):41–66, 1991.

[BDLM09] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To CamelCase or Under_score. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 158–167. IEEE CS, 2009.

[Boe81] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[Bor04] Jürgen Bortz. *Statistik: für Human- und Sozialwissenschaftler*. Springer, sixth edition, 2004.

[Boy77] John Boysen. *Factors Affecting Computer Program Comprehension*. PhD thesis, Iowa State University, 1977.

[Bro78] Ruven Brooks. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 196–201. IEEE CS, 1978.

[DR00] Alastair Dunsmore and Marc Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS 35-2000, Department of Computer Science, University of Strathclyde, 2000.

[FALK11] Janet Feigenspan, Sven Apel, Jörg Liebig, and Christian Kästner. Exploring Software Measures to Assess Program Comprehension. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE CS, 2011. paper 3.

[FBR12] Janet Feigenspan, Don Batory, and Taylor Riché. Is the Derivation of a Model Easier to Understand than the Model Itself? In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 47–52. IEEE CS, 2012.

[FCM$^+$08] Eduardo Figueiredo, Nelio Cacho, Mario Monteiro, Uira Kulesza, Ro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 261–270. ACM Press, 2008.

[Fei09] Janet Feigenspan. Empirical Comparison of FOSD Approaches Regarding Program Comprehension – A Feasibility Study. Master's thesis, University of Magdeburg, 2009.

[FKA^{+}12] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Softw. Eng.*, 2012. DOI: 10.1007/s10664-012-9208-x.

[FKL^{+}12] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring Programming Experience. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 73–82. IEEE CS, 2012.

[FOMMH10] Thomas Fritz, Jingwen Ou, Gail Murphy, and Emerson Murphy-Hill. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 385–394. IEEE CS, 2010.

[FS12] Janet Feigenspan and Norbert Siegmund. Supporting Comprehension Experiments with Human Subjects. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 244–246. IEEE CS, 2012. Tool demo.

[FSF11] Janet Feigenspan, Norbert Siegmund, and Jana Fruth. On the Role of Program Comprehension in Embedded Systems. In *Proc. Workshop Software Reengineering (WSR)*, pages 34–35. GI, 2011. `http://wwwiti.cs.uni-magdeburg.de/iti\_db/publikationen/ps/auto/FeSiFr11.pdf`.

[FSP^{+}11] Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Proc. Int'l Conf. Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75. Institution of Engineering and Technology, 2011.

[HS95] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

[Kit93] Barbara Kitchenham. A Methodology for Evaluating software Engineering Methods and Tools. In H. Rombach, Victor Basili, and Richard Selby, editors, *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, volume 706 of *Lecture Notes in Computer Science*, pages 121–124. Springer, 1993.

[KLM^{+}97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopez, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.

[KTS^{+}09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 611–614. IEEE CS, 2009. Tool demo.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[Pen87] Nancy Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.

[SBA+12] Janet Siegmund, André Brechmann, Sven Apel, Christian Kästner, Jörg Liebig, Thomas Leich, and Gunter Saake. Toward Measuring Program Comprehension with Functional Magnetic Resonance Imaging. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM Press, 2012. New ideas track. Submitted 29.6., Accepted 14.8.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, 1984.

[SFF+11] Michael Stengel, Janet Feigenspan, Mathias Frisch, Christian Kästner, Sven Apel, and Raimund Dachselt. View Infinity: A Zoomable Interface for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1031–1033. ACM Press, 2011.

[Sie12] Janet Siegmund. *Framework for Measuring Program Comprehension*. PhD thesis, University of Magdeburg, Submitted in August 2012.

[SKLA12] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM Press, 2012. Submitted 02.07, Accepted 07.08.

[SM79] Ben Shneiderman and Richard Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *Int'l Journal of Parallel Programming*, 8(3):219–238, 1979.

[SM10] Bonita Sharif and Johnathon Maletic. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 196–205. IEEE CS, 2010.

[Sta84] Thomas Standish. An Essay on Software Reuse. *IEEE Trans. Softw. Eng.*, SE–10(5):494–497, 1984.

[SV95] Teresa Shaft and Iris Vessey. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6(3):286–299, 1995.

[Tia11] Rebecca Tiarks. What Programmers Really Do: An Observational Study. In *Proc. Workshop Software Reengineering (WSR)*, pages 36–37. GI, 2011.

[Tic98] Walter F. Tichy. Should Computer Scientists Experiment More? *Computer*, 31(5):32–40, 1998.

[TLPH95] Walter Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[vMVH97] Anneliese von Mayrhauser, Marie Vans, and Adele Howe. Program Understanding Behaviour during Enhancement of Large-scale Software. *Journal of Software Maintenance: Research and Practice*, 9(5):299–327, 1997.

8 Appendix

8.1 Programming-Experience Questionnaire

In this section, we show the programming-experience questionnaire. In Table 4, we summarize all questions, including the category to which it belongs, the scale for the answer, and the abbreviation, which we use in the remaining tables.

In Table 3, we show the results of an exploratory factor analysis. Our goal was to look for clusters of questions that show a high correlation among themselves. This way, we intend to develop a model that describes programming experience. The next step is to confirm our model with a different sample. To this end, we are currently collecting data of students from different German universities.

Variable	Factor 1	Factor 2	Factor 3	Factor 4	Factor 5
s.C	.723				
s.ObjectOriented	.700			.403	
s.Imperative	.673	.333		.303	
s.Experts	.600	.326			
s.Java	.540		.427		
y.ProgProf		.859			
z.Size		.764			
s.NumLanguages	.335	.489		.403	
s.ClassMates		.449	.403	.424	
s.Functional			.880		
s.Haskell			.879		
e.Courses				.795	
e.Years			-.460	.573	
y.Prog		.493		.554	
s.Logical					.905
s.Prolog					.883

Gray cells denote main factor loadings.

Table 3: Factor loadings of variables in questionnaire.

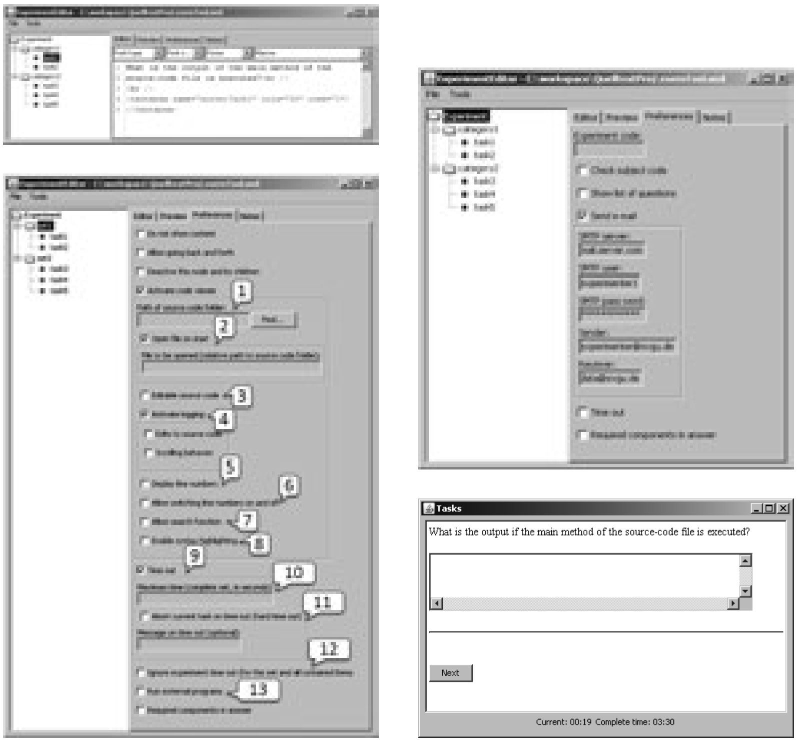
Source	Question	Scale	Abbreviation
Self estimation	On a scale from 1 to 10, how do you estimate your programming experience?	1: very inexperienced to 10: very experienced	s.PE
	How do you estimate your programming experience compared to experts with 20 years of practical experience?	1: very inexperienced to 5: very experienced	s.Experts
	How do you estimate your programming experience compared to your class mates?	1: very inexperienced to 5: very experienced	s.ClassMates
	How experienced are you with the following languages: Java/C/Haskell/Prolog	1: very inexperienced to 5: very experienced	s.Java/s.Prolog/ s.C /s.Haskell
	How many additional languages do you know (medium experience or better)?	Integer	s.NumLanguages
	How experienced are you with the following programming paradigms: functional/imperative/logical/object-oriented programming?	1: very inexperienced to 5: very experienced	s.Functional / s.Imperative / s.Logical / s.ObjectOriented
Years	For how many years have you been programming?	Integer	y.Prog
	For how many years have you been programming for larger software projects, e.g., in a company?	Integer	y.ProgProf
Education	What year did you enroll at university?	Integer	e.Years
	How many courses did you take in which you had to implement source code?	Integer	e.Courses
Size	How large were the professional projects typically?	NA, <900, 900-40000, >40000	z.Size
Other	How old are you?	Integer	o.Age

Integer: Answer is an integer; The abbreviation of each question encodes also the category to which it belongs.

Table 4: Overview of questions to assess programming-experience.

8.2 Prophet

Our tool PROPHET (short for PROgram ComPreHension Experiment Tool) supports experimenters in creating experiments and presenting material to participants. In the experimenter view, experimenters can set up the tasks with HTML. We provide common HTML elements in drop-down lists, as shown in Figure 3.



What is the output if the main method of the source-code file is executed?

Current: 00:19 Complete time: 03:30

Figure 3: Top left: task definition; bottom left: preferences for categories; top right: preferences for the complete experiment; bottom right: task viewer for participants.

Experimenters can also specify settings for a complete experiment. By selecting the check box "Send e-mail", the data of participants are zipped and sent from the specified sender address to the specified receiver address without requiring interaction from participants. Additionally, experimenters can set a time limit for the complete experiment (check box "time out"), after which the experiment is aborted.

Last, we show the view for participants, which shows the task as specified. Participants enter their answer in the text area and navigate forward with the button (labeled "Next").

8.3 Confounding Parameters

As last part of the appendix, we show the confounding parameters we extracted for each journal and conference. For better overview, we divide personal parameters (shown in Table 5 into the categories personal background (i.e., parameters that are defined with the birth and that typically do not change), personal knowledge (i.e., parameters that change only slowly over the course of weeks), and personal circumstances (i.e., parameters that change rapidly, even within minutes). Furthermore, we divide experimental parameters into the categories subject related (i.e., parameters related to the person of the participants, but that occur only because they take part in an experiment), technical (i.e., parameters related to the setting of the experiment), context related (i.e., parameters that typically occur in nearly all experiments), and study-object related (i.e., parameters related to properties of the object under evaluation)[2].

To create a sound experimental design, we recommend traversing this list of parameters and discuss for each parameter whether it is relevant for the experiment and document this process. Additionally, we should also document how we controlled for a confounding parameter. This way, other researchers can consult this documentation when designing experiments and may not trip over neglecting confounding parameters.

Parameter	ESE	TOSEM	TSE	ICPC	ICSE	ESEM	FSE	Sum
Personal background								
Color blindness	0	0	0	1	0	0	0	**1**
Culture	0	0	2	1	0	0	0	**3**
Gender	0	0	3	4	1	0	0	**8**
Intelligence	0	0	2	4	1	0	0	7
Personal knowledge								
Ability	12	2	12	7	5	4	2	**44**
Domain knowledge	3	0	5	4	0	0	0	**12**
Education	8	1	6	15	8	3	0	**41**
Programming experience	24	2	25	23	22	11	5	**112**
Reading time	0	0	0	3	0	1	0	**4**
Personal circumstances								
Attitude toward study object	0	0	1	1	0	0	0	**2**
Familiarity with study object	19	2	17	10	10	12	6	**76**
Familiarity with tools	5	2	9	8	9	1	3	**37**
Fatigue	8	0	5	0	2	5	0	**20**
Motivation	12	0	10	7	3	2	0	**34**
Occupation	0	0	0	3	0	1	0	**4**
Treatment preference	0	0	0	3	1	2	0	**6**

Table 5: *Number of personal confounding parameters mentioned per journal/conference.*

[2]We described each parameter in detail in our PhD thesis [Sie12]

Parameter	ESE	TOSEM	TSE	ICPC	ICSE	ESEM	FSE	Sum
Subject related								
Evaluation apprehension	0	0	1	1	0	0	0	2
Hawthorne effect	9	1	3	2	2	5	0	22
Process conformance	15	1	10	4	5	8	1	44
Study-object coverage	2	0	0	1	0	1	0	4
Ties to persistent memory	0	0	0	1	0	0	0	1
Time pressure	7	0	4	1	0	2	0	14
Visual effort	0	0	0	1	0	0	0	1
Technical								
Data consistency	0	0	1	0	0	0	0	1
Instrumentation	8	0	8	2	0	1	0	19
Mono-method bias	2	0	1	0	0	0	0	4
Mono-operation bias	2	0	1	1	0	0	0	3
Technical problems	0	0	0	2	0	2	0	2
Context related								
Learning effects	15	0	14	16	7	9	4	65
Mortality	0	0	1	0	0	0	1	2
Operationalization of study object	1	1	0	0	0	0	0	2
Ordering	5	0	7	8	2	2	3	27
Rosenthal	10	1	2	3	3	5	0	24
Selection	11	1	6	1	2	2	1	24
Study-object related								
Content of study object	5	1	1	9	0	2	1	19
Language	7	2	14	23	13	7	6	72
Layout of study object	4	0	2	7	0	3	1	17
Size of study object	14	1	19	15	9	6	3	67
Tasks	6	0	6	14	5	4	2	37

Table 6: *Experimental confounding parameters.*

Optimale Integrationsreihenfolgen

Mario Winter

Institut für Informatik
Fachhochschule Köln, Campus Gummersbach
Steinmüllerallee 1
51643 Gummersbach
mario.winter@fh-koeln.de

Abstract: Der Integrationstest prüft das Zusammenspiel der Bausteine eines Softwaresystems. Hierbei bestimmt die gewählte Integrationsstrategie die Reihenfolge, in der die Bausteine integriert und in ihrem Zusammenspiel getestet werden. Zugunsten einer einfacheren Fehlerlokalisierung fügen schrittweise Integrationsstrategien immer nur eine begrenzte Anzahl weiterer Bausteine zur Menge bereits integrierter und integrationsgetesteter Bausteine hinzu. Weist hierbei ein Baustein eine Abhängigkeit zu einem noch nicht in dieser Menge befindlichen Baustein auf, wird letzterer durch einen extra für den Test zu erstellenden Stellvertreter (stub) ersetzt. Wird ein neu hinzugenommener Baustein von noch nicht integrierten Bausteinen genutzt, so sind diese durch Treiber (driver) zu ersetzen. Gegenstand dieses Beitrages ist die Ermittlung einer optimalen Integrationsreihenfolge mit dem Ziel, den Aufwand zur Erstellung von Test-Stellvertretern und -Treibern zu minimieren. Dazu wird die Problemstellung als ganzzahliges Optimierungsproblem formuliert, welches mit Verfahren der dynamischen Programmierung gelöst werden kann. Zwei Experimente zeigen die Leistungsfähigkeit des vorgestellten Ansatzes auf.

1. Einleitung

Testen dient der Bewertung eines Softwareprodukts und dazugehöriger Arbeitsergebnisse mit dem Ziel, deren Anforderungserfüllung und Eignung festzustellen sowie ggf. Defekte zu finden. Dies kann einerseits mit statischen Analysen erfolgen, welche den zu untersuchenden bzw. zu prüfenden Gegenstand (Teil-, Zwischen- oder Endprodukt, im Folgenden als Testobjekt bezeichnet) als solches analysieren und somit auf alle Entwicklungsprodukte wie z. B. Anforderungs- und Entwurfsspezifikationen sowie Programmcode „als Text“ anwendbar sind. Im Gegensatz dazu führt man bei dynamischen Tests – oft einfach als „Testen“ bezeichnet –, „ausführbare“ Testobjekte, d. h. Programmteile, ganze Programme oder Systeme, unter kontrollierten Bedingungen mit dem Ziel aus, die korrekte Umsetzung der Anforderungen nachzuweisen, Fehlerwirkungen aufzudecken und das „Vertrauen“ in das Testobjekt zu erhöhen (vgl. [GTB10]).

Bei dynamischen Tests werden vor der Testausführung für ausgewählte Eingaben anhand der Spezifikation des Prüflings (der „Testbasis“) die erwarteten Ergebnisse ermittelt. Nach der Ausführung wird das jeweilige tatsächliche Ergebnis der Ausführung mit dem vorher ermittelten erwarteten Ergebnis verglichen. Ausführbare Testobjekte im dynamischen Test sind z. B. einzelne Funktionen bzw. Prozeduren, die Instanzen einer Klasse, einzelne Teilsysteme oder das vollständige System.

Je nach der „Konstruktionsphase", in der die Testbasis angelegt wurde, und der Granularität der jeweiligen Testobjekte ergeben sich die korrespondierenden Teststufen Modul- bzw. Komponententest (unit test), Integrationstest, Systemtest und Abnahmetest (vgl. [GTB10]). Der Integrationstest ist als Teststufe zwischen Modul- bzw. Komponententest und Systemtest gelagert. Dabei werden Klassen, Module, Komponenten, Teilsysteme oder auch ganze Systeme zusammengefügt und in ihrem Zusammenspiel geprüft. Zur Vermeidung von Begriffskonflikten wird im Weiteren der allgemeine Begriff „Baustein" verwendet.

Wenn Bausteine zusammenspielen oder aufeinander aufbauen bzw. verweisen, besteht eine Abhängigkeit zwischen den Bausteinen. Jungmayr definiert Abhängigkeiten in seiner Dissertation zum Thema Testbarkeit (testability) folgendermaßen: *A dependency is a directed relationship between two entities where changes in one entity may cause changes in the other (depending) entity* ([Ju03], S. 9). Wichtig dabei ist, dass jede Abhängigkeit immer zwischen genau zwei Bausteinen besteht. Im Rahmen der Abhängigkeit spielen beide Bausteine unterschiedliche Rollen: Es gibt einen abhängigen und einen unabhängigen Baustein, wobei die Abhängigkeit vom abhängigen hin zum unabhängigen Baustein zeigt. Abb. 1 (a) skizziert z. B. eine Abhängigkeit von Baustein `A` zu Baustein `B`, die als `A_B` bezeichnet ist. Der abhängige Baustein `A` benötigt den unabhängigen Baustein `B` für seine korrekte Funktionsweise. Insofern wird der unabhängige Baustein auch oft als „benötigter Baustein" bezeichnet. Abb. 1 (b) zeigt zwei gegenseitig voneinander abhänge Bausteine und damit einen Zyklus im Abhängigkeitsgraphen. Solche Zyklen können auch mehrere Bausteine umfassen und erschweren die Ermittlung einer Integrationsreihenfolge (s. U.).

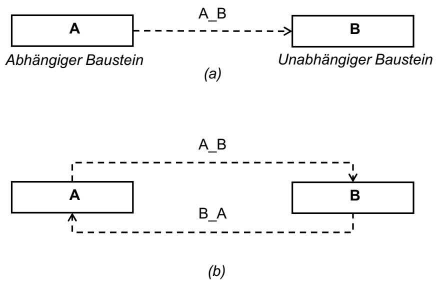


Abbildung 1: Abhängigkeiten zwischen Bausteinen

Der Integrationstest prüft das Zusammenspiel voneinander abhängiger Bausteine eines (Software-) Systems. Dazu werden Testfälle entworfen, deren Eingaben die zu integrierenden Bausteine stimulieren und deren erwartete Ergebnisse die Interaktionen zwischen diesen Bausteinen sowie deren Parameter- und Rückgabewerte umfassen. Hierbei bestimmt die gewählte Integrationsstrategie die Reihenfolge, in der die Bausteine integriert und in ihrem Zusammenspiel getestet werden. Man unterscheidet schrittweise Integrationsstrategien von der sog. „Big-Bang-Strategie“, welche alle Bausteine eines Systems auf einmal integriert und dann (i.d.R. mit Systemtests) testet.

Zugunsten einer einfacheren Fehlerlokalisierung fügen schrittweise Integrationsstrategien immer nur eine begrenzte Anzahl weiterer Bausteine zur Menge bereits integrierter und integrationsgetesteter Bausteine hinzu. Weist hierbei ein Baustein eine Abhängigkeit zu einem noch nicht in dieser Menge befindlichen Baustein auf, wird letzterer durch einen extra für den Test zu erstellenden Stellvertreter (*stub*) ersetzt. Dieser stellt für die im Test geprüfte Nutzung eine rudimentäre Funktionalität sowie ggf. Protokollierungs- und Profiling-Funktionen bereit. Wird ein neu hinzugenommener Baustein von noch nicht integrierten Bausteinen genutzt, so sind letztere durch Treiber (*driver*) zu vertreten, wenn Testfälle für deren Nutzung des aktuell integrierten Bausteins notwendig sind. Dies ist dann der Fall, wenn der zu integrierende Baustein im Rahmen dieser Nutzungen seinerseits weitere Bausteine nutzt.

Schrittweise Integrationsstrategien lassen sich weiter unterteilen in von der Softwarestruktur abhängige und von ihr unabhängige Strategien. Für strukturabhängige Strategien wird oft der Abhängigkeitsgraph herangezogen, dessen Knoten die Bausteine und dessen Kanten ihre Abhängigkeiten darstellen (s. Abb. 1). Prominente Vertreter sind z.B. die Bottom-Up und die Top-Down Strategie, welche von vollkommen unabhängigen (also nur „genutzten“) Bausteinen hin zu vollkommen abhängigen (also nur „nutzenden“) Bausteinen bzw. umgekehrt integrieren. Als Mischformen sind auch die Inside-Out und die Outside-In bzw. Sandwich-Strategie bekannt. Strukturunabhängige Strategien wählen den bzw. die als nächstes zu integrierenden Bausteine z.B. nach deren Risiko, Komplexität oder Verfügbarkeit aus.

Strukturabhängige Strategien sind nur dann unmittelbar z.B. durch eine topologische Sortierung umsetzbar, wenn der Abhängigkeitsgraph keine Zyklen aufweist. Dies ist jedoch bei den interaktions- bzw. nutzungsbasierten Abhängigkeiten in objektorientierten Systemen i.d.R. nicht der Fall ([Ov94][Ku95][BLW03], [Wi00]), so dass „objektorientierte Strategien“ versuchen, z.B. durch das Aufbrechen von Zyklen eine Reihenfolge für die Integration der Klassen zu ermitteln (*class integration test order*, CITO). Allerdings ist dort neben den interaktions- bzw. nutzungsbasierten Abhängigkeiten auch die Vererbung bzw. Generalisierung zu berücksichtigen.

Gegenstand dieses Beitrags ist die Ermittlung einer optimalen Integrationsreihenfolge im Rahmen schrittweiser strukturabhängiger Integrationsstrategien mit dem Ziel, den Aufwand zur Erstellung von Test-Stellvertretern und -Treibern zu minimieren. Im Gegensatz zu anderen Arbeiten wird hierbei keine Heuristik verwendet, sondern das Problem der Ermittlung einer optimalen Integrationsreihenfolge als ganzzahliges Optimierungsproblem modelliert, welches mit Algorithmen der kombinatorischen Optimierung bzw. der dynamischen Programmierung wie z.B. dem Branch- und Bound-Verfahren gelöst werden kann. Das Verfahren wurde in einem kommerziellen Optimierungsprogramm implementiert, um Experimente zu seiner Leistungsfähigkeit durchführen zu können. Der Beitrag stellt das Verfahren selbst sowie die Ergebnisse zweier Experimente vor.

Der Beitrag ist wie folgt strukturiert. Kapitel 2 beschreibt relevante andere Arbeiten aus der zahlreichen vorhandenen Literatur zur Ermittlung von Integrationsreihenfolgen. In Kapitel 3 wird die Problemstellung dann formalisiert, als Optimierungsproblem modelliert und in einem kommerziellen Optimierungs-Werkzeug implementiert. Kapitel 4 fasst die Ergebnisse der beiden Experimente zusammen. Kapitel 5 beendet den Beitrag mit einer Bewertung des Ansatzes und einem Ausblick auf geplante weitere Arbeiten.

2. Andere Arbeiten

Die Arbeiten zur Ermittlung von Integrationsreihenfolgen lassen sich grob in solche klassifizieren, die auf deterministischen graph-basierten Verfahren aufbauen, und solche, die heuristische Ansätze wie z.B. Suchverfahren oder genetische Algorithmen verwenden. Aus Platzgründen werden nur einige wenige repräsentative Arbeiten näher beleuchtet, jeweils stellvertretend für ihre Klasse. Umfassendere Ausführungen hierzu finden sich z.B. in [BLW03], [Ba09], [BP09] oder [Wi12].

2.1 Graph-basierte Ansätze

In einer der ersten Arbeiten zum Integrationstest für objektorientierte Software gibt Overbeck ein Verfahren zur Ermittlung einer Integrationsstrategie auf der Grundlage des Klassendiagramms an ([Ov94]). Das Verfahren orientiert sich primär an den Generalisierungsbeziehungen Top-Down, also von Basisklassen zu abgeleiteten Klassen, und sekundär an den Interaktions- bzw. Nutzungsabhängigkeiten. Die mit diesem Verfahren ermittelten Integrationsstrategien minimieren die Anzahl der für den Test benötigten Testtreiber und -stellvertreter. Probleme, die aus zyklischen Nutzungsbeziehungen resultieren, sollen durch „Aufbrechen“ der Zyklen anhand einer Tiefensuche mit entsprechenden Teststellvertretern behandelt werden. Ebenso durch Aufbrechen zyklischer Abhängigkeiten und nachfolgende topologische Sortierung wird die Integrationsreihenfolge von Kung et Al. ermittelt ([Ku95]).

[Je99] und [Wi00] stellen unabhängig voneinander eine Integrationsstrategie vor, welche die Betrachtung von ganzen Klassen auf deren konstituierende Bausteine verfeinert. Die Knoten der resultierenden Abhängigkeitsgraphen modellieren also Member der Klassen (Methoden, in [Wi00] auch Variablen), die Kanten stellen Aufrufe zwischen Operationen dar, in [Wi00] darüber hinaus auch Redefinitionen von Operationen sowie die Verwendung der Member-Variablen (def, use). Die feingranulare Betrachtung führt dazu, dass viele auf Klassenebene zu beobachtende zyklischen Abhängigkeiten auf der Ebene von Methoden in azyklische Teilgraphen bzw. Wälder zerfallen und somit oft eine „kanonische“ Integrationsreihenfolge gefunden werden kann.

Abdurazik und Offutt beschreiben ein graph-basiertes Verfahren, das sowohl die Knoten als auch die Kanten des Abhängigkeitsgraphen mit Gewichten belegt, welche durch unterschiedliche Code-Metriken ermittelt werden ([AO09]). Unter Berücksichtigung beider Arten von Gewichten erreichen sie in den Experimenten bessere Ergebnisse als vergleichbare Arbeiten.

2.2 Such-basierte Ansätze

Le Hanh et Al. sowie Briand et Al. experimentieren mit evolutionären Algorithmen zur Ermittlung optimaler Integrationsreihenfolgen ([Le01], [BFL02]). Als primäres Ziel- bzw. „Fitness"-Kriterium wird die Kopplungs-Komplexität zwischen Klassen sowie – in [Le01] – auch zwischen Methoden betrachtet. [Le01] verwenden als sekundäres Zielkriterium auch die Komplexität der einzelnen Klassen selbst. Experimentelle Vergleiche verschiedener evolutionärer und graph-basierter Algorithmen ergaben für erstere tw. bessere Ergebnisse, während letztere besser für große Systeme skalierten.

Assunção et Al. betrachten die verschiedenen Arten von Abhängigkeiten separat und formulieren ein Optimierungsproblem mit mehrfacher Zielsetzung ([As11]). Zur Lösung verwenden sie evolutionäre Mehrziel-Optimier-Algorithmen. Experimente mit zwei unterschiedlichen Algorithmen hinsichtlich vier Zielkriterien (Attribute, Methoden, Anzahl unterschiedlicher Typen der Parameter und Rückgabewerte) zeigen die Eignung des Ansatzes anhand vier realer Systeme.

3. Formulierung der Problemstellung und Optimierungsansatz

In diesem Kapitel formulieren wir zunächst die Problemstellung und überführen sie dann in ein ganzzahliges Optimierungsproblem, welches für ein kommerzielles Optimierungswerkzeug implementiert wird.

3.1 Problemformulierung

Seien BM die Menge der zu integrierenden Bausteine mit $|BM| = n$ und $N: BM \rightarrow \{1..n\}$ eine Nummerierung der Bausteine. Wir engen die Problemstellung auf schrittweise Integrationsstrategien ein, welche immer nur genau einen Baustein zur Menge bereits integrierter und integrationsgetesteter Bausteine hinzufügen. Lösungsraum für solche Integrationsstrategien ist somit die Menge aller Permutationen der Elemente von $\{1..n\}$, die im weiteren mit σ bezeichnet wird. Für ein $o \in \sigma$ bedeutet $o(i) = k$ mit $i, k \in \{1..n\}$, dass Baustein i als k-ter Baustein integriert wird.

Weist ein zu integrierender Baustein eine Nutzungs- oder eine Generalisierungs-Abhängigkeit zu einem noch nicht integrierten Baustein auf, ist letzterer durch einen Test-Stellvertreter zu ersetzen. Kann ein neu hinzugenommener Baustein nicht über bereits integrierte Bausteine angesteuert werden, so ist für jede entsprechende Aufruf-Abhängigkeit ein Treiber zu erstellen. Zusätzlich fällt immer ein Treiber für die Ansteuerung des Systems nach dem letzten Integrationsschritt an.

Die Aufwände hierfür lassen sich in drei Matrizen KS, KD und $KG \in Mat_{n,n}(\mathbb{R})$ festhalten. $KS(i, j)$ bzw. $KG(i, j)$ beziffern den Aufwand, wenn für eine Nutzungs- bzw.

Generalisierungs[1]-Abhängigkeit von Baustein *i* zu Baustein *j* ein Stellvertreter als Ersatz für *j* erstellt werden muss. *KD(i, j)* gibt den Aufwand an, für eine Abhängigkeit von *i* nach *j* anstelle des aufrufenden Bausteins *i* einen Treiber einzusetzen. .

Zur Ermittlung des geschätzten Aufwands für die Erstellung von Test-Stellvertretern und –Treibern können unterschiedliche Komplexitätsmetriken wie z.B. bei einer Nutzungsabhängigkeit die Anzahl der unterschiedlichen aus *i* aufgerufenen Methoden von *j* sowie deren Parameter und Rückgabewerte oder bei einer Generalisierungsabhängigkeit die Anzahl der von *i* aus *j* geerbten und verwendeten bzw. ggf. redefinierten Methoden etc. dienen (vgl. z.B. [As11], [BFL02] und [AO09]). Im Weiteren werden o.B.d.A. keine konkreten Aufwände ermittelt, sondern für eine Nutzungs- bzw. Generalisierungs-Abhängigkeit von Baustein *i* zu Baustein *j* die konstanten Werte *KS(i, j)* = *KG(i, j)* = 2 und *KD(j, i)* = 1 angesetzt. Für eine Integrationsreihenfolge $o \in \sigma$ fallen diese Aufwände natürlich nur dann an, wenn $o(i) < o(j)$ ist, Baustein *i* also bzgl. *o* vor Baustein *j* integriert wird.

Die gesamten Kosten *K* für eine bestimmte Integrationsreihenfolge $o \in \sigma$ lassen sich dann berechnen als

$$K(o) = \left(\sum_{i,j=1}^{n} \begin{Bmatrix} KS(i,j) + KG(i,j) + KD(i,j),\ falls\ o(i) < o(j) \\ 0, sonst \end{Bmatrix}\right) + 1 \qquad (1)$$

Für den in Abbildung 2 gezeigten einfachen „generalisierungsfreien“ Abhängigkeitsgraphen sind z.B. *KS(1, 2)* = 2, *KD(1, 2)* = 0 und *KD(2, 1)* = 1. Die „Kosten“ für den ersten Schritt der „Top-Down“ Integrationsreihenfolge o_{TD} = <1, 2, 3, 4, 5, 6, 7> betragen 7 (ein Treiber für Komponente 1 und drei Stellvertreter für die Komponenten 2 bis 4, also 1+3*2=7). Als Gesamtkosten ergeben sich $K(o_{TD}) = 19$. Besser ist die „Bottom-Up“ Reihenfolge o_{BU} = <7, 6, 5, 4, 3, 2, 1> mit den Gesamtkosten $K(o_{BU}) = 10$.

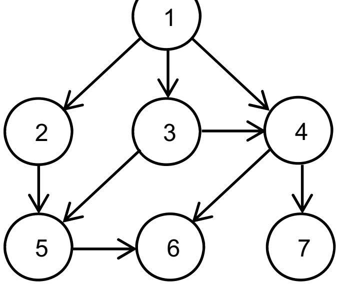


Abbildung 2: Einfacher (azyklischer) Abhängigkeitsgraph

[1] Generalisierungsabhängigkeiten (und Kompositionen) werden i.d.R. gesondert betrachtet, da diese einen azyklischen Graph ergeben (müssen), während dies bei Nutzungsabhängigkeiten (und Assoziationen) nicht der Fall ist.

3.2. Optimierungsansatz

Die Ermittlung einer optimalen Integrationsreihenfolge lässt sich nun folgendermaßen formulieren: Suche ein $o \in \sigma$ für das $K(o)$ ein Minimum ist, kurz: *Minimiere* $K(o)$.

Die vollständige Enumeration aller $n!$ Permutationen verbietet sich schon für recht kleines n. In Anlehnung an Formulierungen bekannter Reihenfolgeprobleme wie z.B. dem Rundreiseproblem (traveling salesman problem, TSP) oder dem Arbeitsplanproblem (job shop scheduling, JSS) wird das Integrationsreihenfolge-Problem als ganzzahliges Optimierungsproblem angesetzt, welches Methoden der dynamischen Programmierung zugänglich ist ([PS82]).

Eine Reihenfolge-Matrix $R \in Mat_{n,n}(\{0,1\})$ eliminiert hierbei die kleiner-Abfrage bzgl. der Reihenfolge bei der Berechnung der Integrationskosten in (1). Ist $R(i, j) = 1$, dann wird Baustein i vor Baustein j integriert. Darüber hinaus fassen wir die Einzelkosten-Matrizen KS, KD und KG in einer einzigen Kostenmatrix $C \in Mat_{n,n}(\mathbb{R})$ zusammen, mit $C(i, j) = KS(i, j) + KG(i, j) + KD(i, j)$. Der konstante Faktor eins in (1) wird vernachlässigt, da er bei allen Reihenfolgen für den Treiber des Systems anfällt.

Nun müssen noch die Bedingungen angegeben werden, unter denen die Reihenfolge-Matrix R tatsächlich eine Permutation der n Bausteine widerspiegelt. Zunächst ist für je zwei Bausteine i und j entweder i vor j oder aber j vor i zu integrieren:

$$\forall i,j \in \{1..n\}, i < j: R(i,j) + R(j,i) = 1 \qquad (2)$$

Für die Permutation $o \in \sigma$ muss dann gelten (vgl. [PS82] S. 311):

$$\forall i,j \in \{1..n\}, i \neq j: \big(o(i) - o(j) + 1 \ \leq n \cdot R(j,i)\big) \qquad (3)$$

Die Bedingung in (3) sagt zunächst aus, dass sich die in R kodierte Reihenfolge auch in o wiederfindet. Darüber hinaus erzwingt (3), dass keine Differenz je zweier Werte in o größer als $n - 1$ ist, o also genau alle Werte von 1 bis n umfasst. Es ergibt sich die folgende Optimierungsvorschrift:

$$Min \sum_{i,j=1}^{n} C(i,j) \cdot R(i,j) \quad \text{s.t. } (2) \wedge (3) \qquad (4)$$

3.3 Implementierung

Im IBM ILOG CPLEX Optimization Studio ([IBM12]) lässt sich das obige Optimierungsproblem in Form der in Abbildung 3 gezeigten Zielfunktion und Beschränkungsmenge implementieren. Die Variable `n` gibt die Kardinalität der Bausteinmenge an, die Variable `edges` mit den Komponenten `<i,j>` umfasst die Kanten des in Listenform kodierten Abhängigkeitsgraphen. Für jede Kante `<i,j>` gibt der Eintrag `cost[<i,j>]` die Einzelkosten bei der Integration von Baustein `i` vor Baustein `j` an. `R` ist die Reihenfolgematrix, `o` die aktuelle Permutation.

```
// Objective
minimize sum (<i,j> in edges) cost[<i,j>]*R[i,j];
subject to {

        forall ( i in 1..n )
          forall ( j in i+1..n ) {
            R[i,j] + R[j,i] == 1;
          }

        forall ( i in 1..n )
          forall ( j in 1..n ) {
            if ( i != j ) {
              [i] - o[j] + 1 <= n * R[j,i];
        }
```

Abbildung 3: Model-Code des Ziels und der Beschränkungen für den Solver

4. Erste Experimente

In diesem Abschnitt werden zwei Experimente beschrieben, die mit der in Abbildung 3 gezeigten Implementierung des Optimierungsmodells durchgeführt wurden.

4.1 Project Planning Tool

Das Beispielsystem dieses Experiments stammt aus der Dissertation von Overbeck ([Ov94]). Der in Abbildung 4 gezeigte Abhängigkeitsgraph zeigt 8 Bausteine und ihre 26 Nutzungsabhängigkeiten. Zusätzlich ist die Nummerierung der Bausteine angegeben.

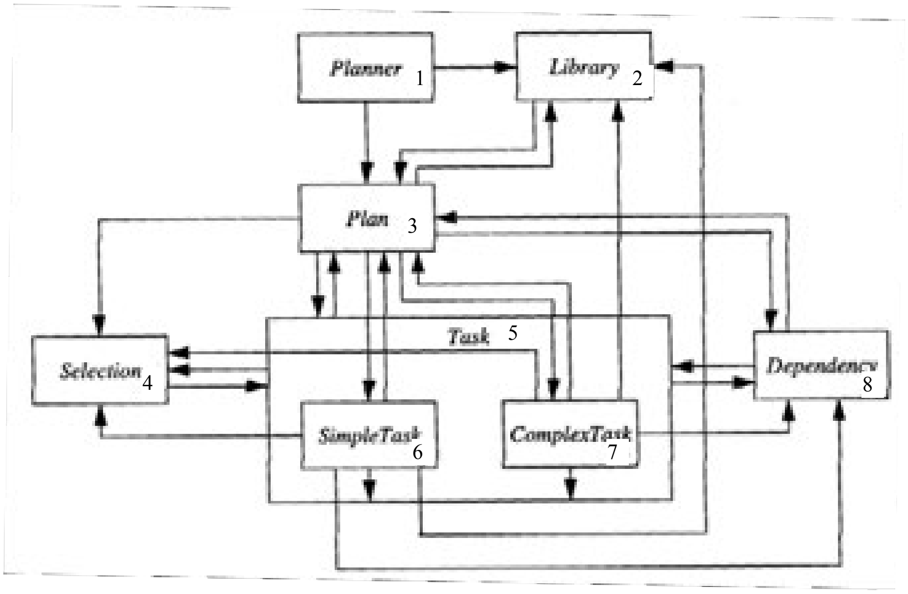


Abbildung 4: Project Planning Tool (PPT, aus [Ov94])

Die beiden Bausteine `SimpleTask` und `ComplexTask` stehen in einer Generalisierungs-abhängigkeit zum Baustein `Task`, was in Abbildung 4 durch das „Enthaltensein“ in sowie Nutzungsbeziehungen zu dem Baustein-Rahmen des Letzteren symbolisiert ist. Die Kodierung des Problems zu seiner Optimierung umfasst also 26 Kanten mit den jeweiligen Kostenfaktoren.

Abbildung 5 zeigt das Setup-Log des Solvers nach der Modell-Kompilierung. Das Modell führte zu einem Gleichungssystem mit 72 Variablen und 85 Beschränkungen.

```
! ----------------------------------------------------------------------------
! Minimization problem - 72 variables, 85 constraints
! Initial process time : 0,00s (0,00s extraction + 0,00s propagation)
!  . Log search space  : 88,0 (before), 88,0 (after)
!  . Memory usage      : 411,5 kB (before), 427,5 kB (after)
! Using parallel search with 2 workers.
! ----------------------------------------------------------------------------
```

Abbildung 5: PPT - Setup-Log des Solvers

Abbildung 6 zeigt im Ergebnis-Log des Solvers, dass zwei Lösungen gefunden wurden, deren Beste den Wert 32 liefert. Dafür wurden 88.115 Zweige in 2,21 Sekunden eruiert.

```
! ----------------------------------------------------------------------------
! Search terminated normally, 2 solutions found.
! Best objective         : 32 (optimal - effective tol. is 0)
! Number of branches     : 88.115
! Number of fails        : 43.844
! Total memory usage     : 1,8 MB (1,5 MB CP Optimizer + 0,2 MB Concert)
! Time spent in solve    : 2,21s (2,21s engine + 0,00s extraction)
! Search speed (br. / s) : 39.713,8
! ---------------------------------------------------------------------------
```

Abbildung 6: PPT Ergebnis-Log des Solvers

Die optimale Lösung des Solvers mit Kosten von 32 ist in Abbildung 7 gelistet. In der ersten Zeile wird der erreichte Zielwert angegeben. Darunter sind die Reihenfolge-Matrix *R* und die optimale Reihenfolge *o* der acht Bausteine aufgeführt. Hierbei bedeutet *o*(5) = 1, dass die Integration mit Baustein 5 (`Task`) beginnt. Zuletzt wird Baustein 7 (`ComplexTask`) integriert (*o*(7) = 8). Im Vergleich dazu erfordert die von Overbeck angegebene Reihenfolge <8, 5, 6, 2, 1, 7, 4, 3> die Kosten von 34 ([Ov94], S. 110ff.).

```
// solution with objective 32
R =  [[0 0 0 0 0 1 1 0]
       [1 0 1 0 0 1 1 0]
       [1 0 0 0 0 1 1 0]
       [1 1 1 0 0 1 1 1]
       [1 1 1 1 0 1 1 1]
       [0 0 0 0 0 0 1 0]
       [0 0 0 0 0 0 0 0]
       [1 1 1 0 0 1 1 0]];
o = [6 4 5 2 1 7 8 3];
```

Abbildung 7: PPT Lösung des Solvers

4.2 ATM Simulator

Das zweite Experiment verwendet die in Abbildung 8 dargestellten Abhängigkeiten der Automated Teller Machine (ATM) Simulation aus [BFL02], App. A, S. 38ff. Das System umfasst 21 Klassen, deren Abhängigkeiten durch Generalisierungs- bzw. Vererbungsbeziehungen (*inheritance*) sowie durch Kompositions- (*composition*), Assoziations- und Nutzungsbeziehungen zustande kommen.

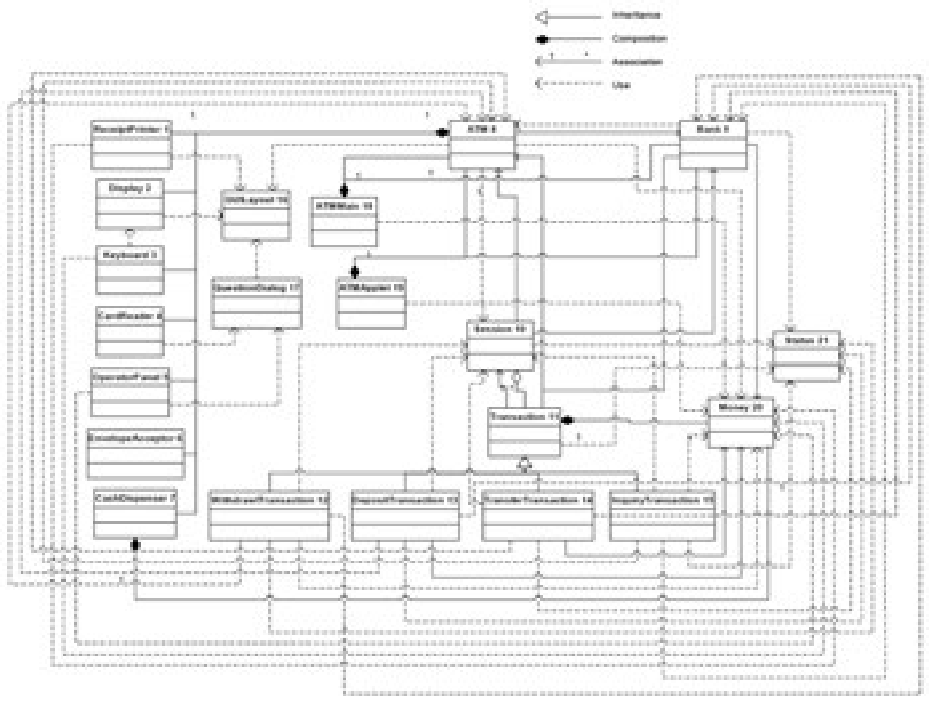


Abbildung 8: ATM-Simulator (ATM, aus [BFL02])

Bei den von Briand et Al. in [BFL02] beschriebenen Experimenten mit genetischen Algorithmen wurde festgelegt, dass die durch die Generalisierungs- sowie durch Kompositionsbeziehungen vorgegebenen Präzedenzen nicht durchbrochen werden dürfen. Im Optimierungsmodell wurden die entsprechenden Kanten daher mit den Maximalkosten belegt. Die übrigen Abhängigkeits-Kanten wurden in diesem Experiment mit den Werten der in [BFL02], A.3 auf S. 41 angegebenen Attribut-Kopplungsmatrix gewichtet.

Abbildung 9 und 10 zeigen das Setup- und das Ergebnis-Log des Solvers für die optimale Integrationsreihenfolge unter der Metrik *Attribut-Kopplung* (A, vgl. [BFL02]). In 145.943 Läufen wurden 13 optimale Reihenfolgen mit dem Zielfunktionswert 39 gefunden (aus max. 21! ≈ $5{,}1*10^{19}$ Möglichkeiten). Dieser wird auch von allen 10 in [BFL02] angegebenen Reihenfolgen erreicht. Eine der optimalen Reihenfolgen ist <20, 16, 17, 21, 5, 7, 2, 1, 6, 3, 4, 8, 9, 11, 10, 19, 18, 15, 13, 12, 14>.

```
! ----------------------------------------------------------------------------
! Minimization problem - 462 variables, 631 constraints
! Initial process time : 0,00s (0,00s extraction + 0,00s propagation)
!  . Log search space  : 533,2 (before), 533,2 (after)
!  . Memory usage      : 0,9 MB (before), 1,1 MB (after)
! Using parallel search with 2 workers.
! ----------------------------------------------------------------------------
```

Abbildung 9: ATM Setup-Log des Solvers (Attribute Coupling)

```
! ----------------------------------------------------------------------------
! Search terminated normally, 13 solutions found.
! Best objective         : 39 (optimal - effective tol. is 0)
! Number of branches     : 145.943
! Number of fails        : 68.034
! Total memory usage     : 5,3 MB (5,0 MB CP Optimizer + 0,3 MB Concert)
! Time spent in solve    : 10,39s (10,39s engine + 0,00s extraction)
! Search speed (br. / s) : 14.045,6
! ----------------------------------------------------------------------------
```

Abbildung 10: ATM Ergebnis-Log des Solvers (Attribute Coupling)

Die Ergebnisse zeigen, dass das einfache Optimierungsmodell aus (4) in der Lage ist, die mit den in der Literatur angegebenen Verfahren ermittelten Integrations-Reihenfolgen als optimal zu bestätigen oder sogar zu verbessern.

5. Zusammenfassung, Bewertung und Ausblick

Der vorgestellte Ansatz ermittelt eine optimale Integrationsreihenfolge mit dem Ziel, den Aufwand zur Erstellung von Test-Stellvertretern und -Treibern zu minimieren. Dazu wird die Problemstellung als ganzzahliges Optimierungsproblem formuliert, welches mit Verfahren der dynamischen Programmierung gelöst werden kann. Zwei Experimente zeigten die Leistungsfähigkeit des vorgestellten Ansatzes auf.

Die ganzzahlige Optimierung ist ein NP-hartes Problem, so dass die z.Zt. bekannten Lösungsverfahren im schlechtesten Fall exponentielle Laufzeit aufweisen ([PS82]). Die aktuellen Algorithmen zur Lösung konkreter Problemstellungen haben jedoch aufgrund der verwendeten Verfahren der linearen Programmierung in Verbindung z.B. mit Cutting-Plane-Algorithmen und Branch&Bound-Verfahren ([PS82]) zur Bestimmung ganzzahliger Lösungen in der Praxis oft eine erstaunlich hohe Effizienz ([IBM12]). Das Verfahren kann somit zumindest für kleine bis mittlere Systeme unmittelbar praktisch eingesetzt werden. Es ermöglicht aber auch die Ermittlung optimaler "Referenzreihenfolgen", anhand derer heuristische Verfahren bewertet und evaluiert werden können.

Unsere aktuellen Arbeiten loten die Grenzen des Ansatzes hinsichtlich Größe und Komplexität der zu integrierenden Software-(Teil-)Systeme aus. Darüber hinaus experimentieren wir mit unterschiedlichen Komplexitätsmetriken zur Ermittlung der Kanten-Gewichte, also des geschätzten „Aufwands“ für die Erstellung von Test-Stellvertretern und –Treibern.

Literaturverzeichnis

[AO09] Abdurazik, A. & Offutt, J.: Using Coupling-Based Weights for the Class Integration and Test Order Problem. Comput. J., Oxford University Press, 2009; S. 557-570.

[As11] Assunção, W. K. G.; Colanzi, T. E.; Pozo, A. T. R. & Vergilio, S. R.: Establishing integration test orders of classes with several coupling measures. Proc. 13th Conf. on Genetic and evolutionary computation, ACM, 2011; S. 1867-1874.

[Ba09] Bansal, P.; Sabharwal, S. & Sidhu, P.: An investigation of strategies for finding test order during integration testing of object-oriented applications. Proc. Int. Conf. on Methods and Models in Computer Science (ICM2CS 09), 2009; S. 1-8.

[BFL02] Briand, L. C.; Feng, J. & Labiche, Y.: Experimenting with Genetic Algorithms to Devise Optimal Integration Test Orders. Technical Report SCE-02-03, Carleton University, 2002.

[BLW03] Briand, L. C.; Labiche, Y. & Wang, Y.: An Investigation of Graph-Based Class Integration Test Order Strategies. IEEE Transactions on Software Engineering, Vol. 29, IEEE Computer Society, 2003; S. 594-607.

[BP09] Borner, L. & Paech, B.: Integration Test Order Strategies to Consider Test Focus and Simulation Effort. Proc. 1st Int. Conf. on Advances in System Testing and Validation Lifecycle (VALID 09), 2009; S. 80-85.

[GTB10] ISTQB/GTB Standardglossar der Testbegriffe, Version 2.1, Deutsch/Englisch, ISTQB/GTB, 2010.

[IBM12] Web-Seiten des IBM ILOG CPLEX Optimization Studio *http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/* (zuletzt besucht am 10.10.2012)

[Je99] Jeron, T.; Jezequel, J.-M.; Le Traon, Y. & Morel, P.: Efficient strategies for integration and regression testing of OO systems. Proc. 10th Int. Symp. onSoftware Reliability Engineering, IEEE, 1999; S. 260-269.

[Ju03] Jungmayr, S.: Improving Testability of Object-Oriented Systems. Dissertation, Fernuniversität Hagen, 2003.

[Ku95] Kung, D., Gao, J., Hsia, P., Toyoshima, Y. & Chen, C.: A test strategy for object-oriented programs. Proc. 19th Computer Software and Applications Conf. (COMPSAC 95), IEEE Computer Society Press, 1995; S. 239-244.

[Le01] Le Hanh, V.; Akif, K.; Le Traon, Y. & Jézéque, J.-M.: Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies. In: Knudsen, J. (Ed.): Proc. ECOOP 2001, LNCS 2072, Springer, Berlin & Heidelberg, 2001; S. 381-401.

[Ov94] Overbeck, J.: Integration Testing for Object-Oriented Software. PhD Dissertation, Vienna University of Technology, 1994.

[PS82] Papadimitriou, C. H. & Steiglitz, K.: Combinatorial optimization - Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, N.J., 1982.

[Ve10] Vergilio, S.; Pozo, A.; Arias, J.; da Veiga Cabral, R. & Nobre, T.: Multi-objective optimization algorithms applied to the class integration and test order problem. International Journal on Software Tools for Technology Transfer (STTT), Springer Berlin & Heidelberg, 2010; S. 1-15.

[Wi00] Winter, M.: Ein interaktionsbasiertes Modell für den objektorientierten Integrations- und Regressionstest. Informatik - Forschung und Entwicklung, Bd. 15, Springer Berlin / Heidelberg, 2000; S. 121-132.

[Wi12] Winter, M.; Ekssir-Monfared, M.; Sneed, H.M.; Seidl, R.; Borner; L.: Der Integrationstest – Von Entwurf und Architektur zur Komponenten- und Systemintegration. Hanser Verlag, München, 2012.

Temporal Reconfiguration Plans for Self-Adaptive Systems*

Steffen Ziegert and Heike Wehrheim

Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
{steffen.ziegert, wehrheim}@uni-paderborn.de

Abstract: Self-adaptive systems have the ability to autonomously reconfigure their structure in order to meet specific goals. Such systems often include a *planning* component, computing plans of reconfiguration steps. However, despite the fact that reconfigurations take time in reality, most planning approaches for self-adaptive systems are non-temporal.

In this paper, we present a model-based approach to the generation of *temporal* reconfiguration plans. Besides allowing for durative reconfigurations, our technique also neatly solves concurrency issues arising in such a temporal setting. This provides the modeller with a clear and easy-to-use basis for modelling while at the same time giving an automatic method for plan construction.

1 Introduction

The development of self-adaptive systems belongs to the most complex tasks in software engineering today and requires a rigorous model-driven approach. Self-adaptive systems possess the ability to accommodate to changes in their environment at runtime. Such changes may include the failure of system components, the activities of other agents in the environment, or changing user demands.

Self-adapting to a resulting new situation, calls for a number of reconfigurations that have to be executed. In general, such reconfigurations do not have to be restricted to components' internal states, but may include changes of the system's architecture like the creation and deletion of software component instances or communication links between them.

Systems that decide when and how to adapt their architecture, are said to have a self-organizing [GMK02] or self-managing [BCDW04] architecture. Kramer and Magee [KM07] defined a reference architectural model with three layers for the development of such systems. It consists of a goal management, a change management, and a component control layer. The goal management layer accomplishes time-consuming tasks, like the computation of reconfiguration plans for given goal specifications. The resulting reconfiguration plans state how to adapt to the environment, i.e. which reconfigurations to apply,

*This work was developed in the course of the Collaborative Research Centre 614 "Self-optimizing Concepts and Structures in Mechanical Engineering" and funded by the German Research Foundation (DFG).

and when to execute them. The second layer, the change management layer, provides the capabilities to adapt the system's architecture. The component control layer is the bottom layer and accomplishes the basic tasks of the system by providing the implementation of primitive features. In this paper, we are specifically concerned with the top layer, i.e. with the creation of *reconfiguration plans*.

We employ a model-based approach to the design of self-adaptive systems. Reconfigurations concern the architecture of the system. System models are given in MECHATRONICUML [BBB+12], a UML profile for the development of self-adaptive mechatronic systems. Here, (initial as well as other) system architectures are modelled as *graphs*, and reconfigurations as story patterns [FNTZ98], a formalism based on *graph transformations*, which schematically define how an architecture configuration can be transformed into a new one. Graph transformations are frequently used for specifying the reconfigurations of self-adaptive systems, e.g. [LM98, WF02], and enable verification techniques to be applied [BBG+06, SWW11]. However, only few apply planning techniques on these models, e.g. [EW11], to actually generate reconfiguration plans that achieve the system's goals. However, this is needed for the entire reconfiguration process to happen autonomously, cf. [BCDW04].

The reconfigurations of a system architecture consume *time* and might be carried out *in parallel*. In fact, in some application domains it might be counterintuitive to execute the reconfigurations sequentially, because they refer to highly independent components (or even different agents in multi-agent systems). However, most current planning approaches to architectural reconfiguration, do not support time, and consequently only generate non-temporal plans. The only one that does, is [TK11], an approach that also builds on graph transformation rules for modelling reconfigurations. However, their approach to assign annotations like «`at_start`» and «`at_end`» as stereotypes to the elements of rules is cumbersome from a modelling perspective: possible interferences force the rules to be precisely formulated.

The approach we follow here is an extension of [TK11]. We solve planning tasks by translating (parts of) our models into the Planning Domain Definition Language (PDDL) and feeding them into an off-the-shelf planning system, like SGPlan$_6$ [CWH06]. However, our solution renounces from assigning start and end annotations. Instead, we apply a *locking mechanism* that is based on the idea of restricting read or write access to elements, when they are in use by a reconfiguration. This is similar to the concurrency control methods implemented by database management systems. Unintended interferences, e.g. the deinstantiation and use of a software component at the same time, are thus avoided, without the need for a modeller to specify this. Apart from a model-based approach for planning time-consuming architectural reconfigurations, our approach can also be seen as a knowledge engineering contribution to the PDDL community.

2 Application Scenario

Our running example is based on the RailCab project that is developed at the University of Paderborn. The RailCab project consists of autonomous, driverless shuttles, called

RailCabs, that operate on a railway system. Each RailCab has an individual goal, e.g. transporting passengers or goods to a specified target station. An important feature of the project is the RailCabs' ability to drive in convoy mode, i.e. RailCabs can minimize the air gap between each other to safe energy. To safely operate in a convoy, acceleration and braking has to be coordinated and managed between convoy members. The instantiation and deinstantiation of software components for the communication and coordination of RailCabs is one example for the high dynamics of the system's communication structure. For the purpose of this paper, we keep the application example small and do not cover every aspect of the system's communication structure.

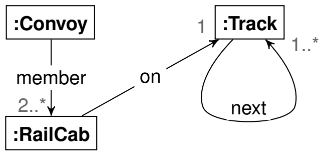


Figure 1: Class diagram of our (simplified) RailCab scenario

We begin with the specification of the system in MECHATRONICUML by first modelling its structure as a class diagram (see Figure 1). The railway system consists of track segments that are connected to each other via `next` links. A RailCab that operates in the system can occupy such a track segment. Furthermore, RailCabs can coordinate with other RailCabs to form a convoy. Such an active convoy operation is represented by an instance of the `Convoy` type. A `Convoy` instance has a `member` link to each participating RailCab.

In MECHATRONICUML, the behaviour of components – more precisely their roles – is modelled with real-time state charts, which allow the definition of communication and timing constraints. Structural reconfigurations, e.g. the instantiation of a convoy, are modelled with story patterns, an extension of UML object diagrams. Since our contribution deals with modelling the structural reconfiguration of self-adaptive systems, the internal behaviour of components is not considered here.

To model reconfiguration actions of our example system, we use story patterns that are typed over the class diagram shown in Figure 1. Story patterns have a formal semantics[1] based on (typed) graph transformation systems [EHK+97]. A graph transformation system consists of a graph representing the initial configuration of the system and a set of rules. Each rule consists of a pair of graphs, called left-hand side (LHS) and right-hand side (RHS), that schematically define how the graph representing the system's configuration can be transformed into new configurations. Elements that are specified in both graphs are preserved, other elements are deleted (if specified in the LHS only) or created (if specified in the RHS only). Syntactically, a story pattern represents such a rule by integrating the LHS and RHS into one graph and using stereotypes to indicate elements that are only present in the LHS or in the RHS [FNTZ98]. The graph-like representation allows not only for an intuitive modelling but also harmonizes with modern approaches for the development of self-optimizing mechatronical systems, e.g. [THHO08].

Figure 2 provides an example of a reconfiguration which takes 4 time units: a RailCab

[1]Story patterns follow the single pushout approach to graph transformation.

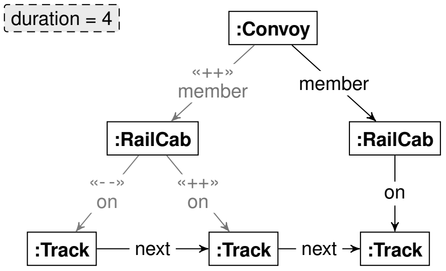


Figure 2: Story pattern `joinConvoy`

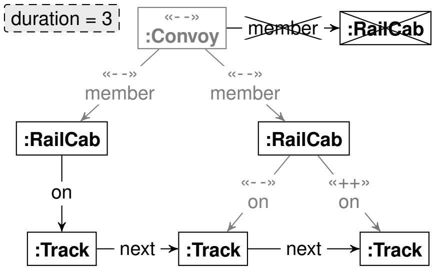


Figure 3: Story pattern `breakConvoy`

joining a convoy of RailCabs. Objects and links that are being created or deleted by the application of the story pattern are labelled with the stereotypes «++» and «- -», respectively. The story pattern specifies the creation of a `member` link representing the RailCab's participation in the convoy operation simultaneously with its movement to the next track segment. The story pattern can be executed to transform the state graph into a new configuration if it contains a subgraph that *matches* the LHS of the story pattern.

Our modelling formalism also allows to express that certain objects or links are not permitted to appear in the current state graph. See for example the story pattern given in Figure 3. The crossed out `RailCab` object and the link connecting it to the `Convoy` object are not allowed to appear in the state graph. Such a restriction to the applicability of a story pattern is called a negative application condition (NAC).

The locking mechanism we developed is restricted to specific kinds of NACs. We use the terms *forbidden link* and *forbidden pair* to refer to these kinds of NACs. A forbidden link denotes a NAC that consists of a single link only. A forbidden pair denotes a NAC that consists of a single object and a link connecting this object to the LHS (as in Figure 3). A *connecting object* denotes an object within a LHS that is connected to a forbidden pair (e.g. the `Convoy` object in Figure 3).

In addition to the story patterns that define possible transformations, we need an initial configuration and a goal specification to feed the planning system with. A goal specification is a partly specified configuration that can be modelled as an ordinary object graph. Goal specifications are either generated from user input or predefined by the system designer. When the self-adapting system is in operation, initial configurations for the planning subsystem are generated from actual runtime states of the system.

3 Concurrent Execution

If we allow only sequential execution of reconfigurations, a plan is a sequence of graph transformations (or – in terms of PDDL – a sequence of ground actions). Considering reconfigurations as durative does not change this as long as we do not allow a concurrent execution of two reconfigurations. If we allow concurrency, the application intervals of two graph transformations can overlap and a plan thus is a set of tuples of points in time and graph transformations. The questions that arise are: does the concurrent execution of two graph transformations result in any conflicts, and how can such a concurrent execution be avoided? An intuitive approach is to assume that the applicability check happens (in zero time) at the beginning of the reconfiguration and the actual change at its end. This is also assumed as default, i.e. when no further annotation is specified, in [TK11].

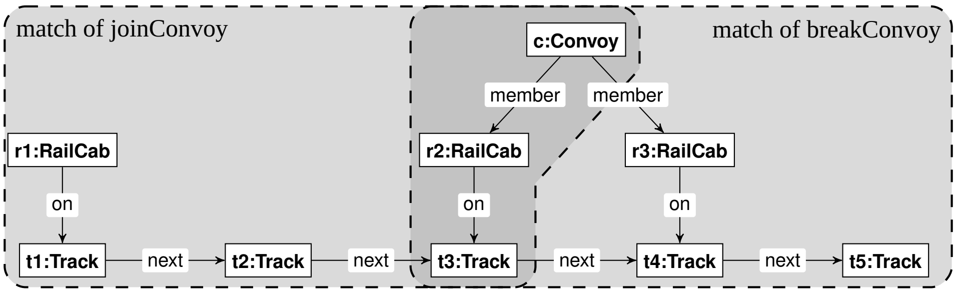


Figure 4: Configuration of the system in which two story patterns can be applied

Unfortunately, such an approach is not suitable for many situations. Consider for instance the state graph given in Figure 4 and the application of the story patterns `joinConvoy` and `breakConvoy` given in Figure 2 and 3. Each of the rules has only one match in the state graph. In Figure 4 they are highlighted by a dashed box. Let us assume that one of the reconfigurations, e.g. `breakConvoy`, is currently being applied. This means, its condition has already been checked but the alteration of the configuration has not yet been executed. The execution of a reconfiguration of RailCab `r1` joining the convoy makes no sense in this situation and should not be allowed because the convoy will be deinstantiated by `breakConvoy`. The problem is that the configuration is in the process of being changed, but this is not reflected in the intermediate state graph. Checking the applicability at the beginning of a reconfiguration and executing the alteration at its end is ineligible as a general solution. To solve this problem, we add information about the execution of the story patterns into the configuration. This can be seen as locking access to the elements of the state graph. Whether a second reconfiguration is allowed to be applied concurrently, can thus be checked by testing for the locks.

There are four cases of concurrency between two reconfigurations that might lead to conflicts. In all of them, an element is concurrently being read (required or forbidden) by a reconfiguration and being written (deleted or created) by another reconfiguration. They differ only in their beginning and ending times.

A schematic overview of the four conflict cases for the story patterns `joinConvoy` and `breakConvoy` is shown in Figure 5. The aforementioned example, in which the execution of `breakConvoy` begun before the `joinConvoy` was applied, corresponds to cases `a` and `b`. The reconfiguration removes the connection of RailCab `r2` to the convoy which is required by `joinConvoy`. In case `a`, `breakConvoy` ends first causing the convoy to be deinstantiated and the alteration of `joinConvoy` not to work anymore. In case `b`, the execution of `joinConvoy` ends first and the alteration of `breakConvoy` is still possible on the new state graph created by executing `joinConvoy`. However, such a concurrent

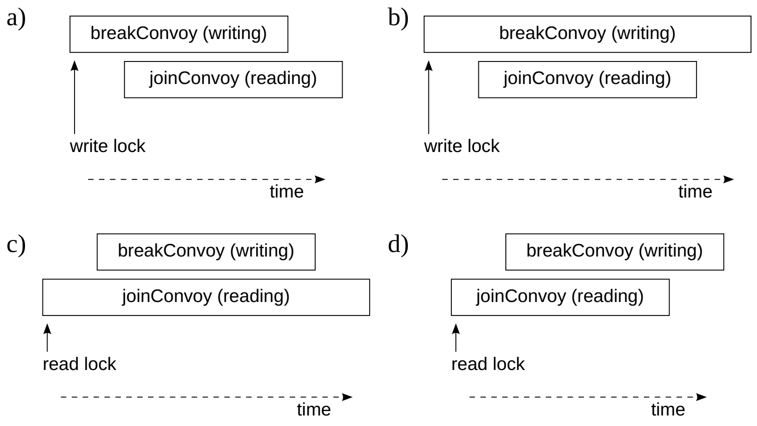


Figure 5: Four conflict cases that have to be solved by locking

execution of `joinConvoy` and `breakConvoy` is not intuitive: although `r1` joined a convoy and was not involved in the `breakConvoy` reconfiguration, there is no convoy that `r1` can be member of after the execution of both reconfigurations. From the perspective of `breakConvoy`, `joinConvoy` did not take the pending alteration of `breakConvoy` into account. Our solution to this problem encodes information about the deinstantiation of the convoy into the configuration by acquiring a write lock of the `Convoy` object when the `breakConvoy` reconfiguration starts and releasing the lock when the reconfiguration ends.

Although conflict cases `c` and `d` differ in their ordering of the reconfigurations' starting points, they are essentially caused by the same reason: either the alteration of `breakConvoy` causes the convoy to be deinstantiated while `r1` joins the convoy (case `c`) or `joinConvoy` does not take the pending alteration of `breakConvoy` into account (case `d`). These situations, however, require a different solution due to their different ordering of starting points. Since `joinConvoy` starts first, it cannot check the information about the deinstantiation of the convoy that `breakConvoy` is going to encode into the configuration by acquiring a write lock. Therefore, `joinConvoy` itself encodes into the configuration that it requires the `Convoy` object by acquiring a read lock of the `Convoy` object. Our solution approach essentially uses write locks on parts of the system's configuration to cope with cases `a` and `b` and read locks to cope with cases `c` and `d`.

4 Translation into PDDL

The planning technique we employ on our application scenario translates the story patterns into PDDL and uses an off-the-shelf planner to compute a reconfiguration plan that transforms a given initial configuration into a target configuration. We incorporated the locking mechanism explained in the last section directly into the translation, thus allowing for the computation of parallel plans that are guaranteed to be free of conflicts. The parallel plans contain precise timing information for the application of the reconfigurations.

Since version 2.1 of PDDL [FL03] it is possible to specify durative actions that allow for concurrent execution. While PDDL's semantics already gives precise timing information for a resulting plan, its semantics regarding concurrent execution is too liberal from the perspective of our model. PDDL's semantics does not sufficiently constrain what kind of concurrent execution is allowed, thus burdening the designer of the planning domain with the complicated and error-prone definition of additional predicates to safely control whether a concurrent execution is allowed. Our translation scheme implements a suitable concurrency control by generating locking predicates in accordance with the locking mechanism we outline in the last section.

In PDDL, a planning task consists of a domain and a problem file. The domain defines action schemata, as well as types and predicates that can be used within action schemata. An action schema consists of a list of parameters, a precondition, and an effect. In the precondition, a list of literals that are required for applying the action can be specified. Similarly, the effect of an action specifies a list of literals that are obtained when the action is applied. An action is instantiated – in the context of PDDL this is called *grounding* – by substituting the parameters with objects that are defined in the problem file, thus transforming the first-order literals into a set of atomic facts that do not contain any free variables. In addition to the objects in the world, the problem file also defines the initial state and a goal specification, both in terms of a set of atomic facts.

Durative actions split the literals used in their precondition and effect into different sets according to their time of evaluation. Literals can be asserted `at_start`, `over_all`, and `at_end` when used in the precondition and be effective `at_start` and `at_end` when used in the effect. While `at_start` and `at_end` refer to the starting and ending timestamp of an action, `over_all` refers to the (open) interval during the action's execution.

The translation schema we describe here explains the construction of a PDDL domain file out of a given MECHATRONICUML model, i.e. a class diagram and a set of story patterns. Roughly spoken, the class diagram yields the declarations (of types and predicates) in the domain file and each story pattern yields an action schema. In doing so, the LHS of a story pattern constitutes the precondition of the action and the RHS its effect.

Class diagram. The translation process begins with the declaration of types, predicates, and functions. All declarations can be deduced from the class diagram. Listing 1 shows the generated declarations for our application scenario. Each type in the class diagram gives rise to a type in the type declaration section `:types`. Every type is a subtype of `Object`, the root of the type hierarchy. We use a predicate `active` for the supertype `Object` stating whether a given object exists, because PDDL does not allow for object creation or deletion. Each link in the class diagram is translated into a predicate that is parameterized by its source and target type. In PDDL, parameters are denoted by variable names followed by their types, e.g. `?track - Track`.

For now, we skip the declaration of the locks. We first cover the translation of story patterns by the example of `joinConvoy` without addressing the locking functionality, then give an overview of the modifications to realize the locks, and then stepwise extend the translation.

Listing 1: Generated declaration of types, predicates, and functions

```
(:types  Convoy - Object  RailCab - Object  Track - Object)
(:predicates
    (active  ?object - Object)
    (member_Convoy_RailCab  ?convoy - Convoy  ?railcab - RailCab)
    (on_RailCab_Track  ?railcab - RailCab  ?track - Track)
    (next_Track_Track  ?track1 - Track  ?track2 - Track)
)
```

Story patterns. Listing 2 shows the translation of the story pattern `joinConvoy`. All the conditions have to hold `at_start`, i.e. at the beginning of the story pattern's execution. The changes, i.e. the creation and deletion of objects or links, happen at the end of its execution.

Listing 2: Generated durative action `joinConvoy`

```
1  (:durative-action joinConvoy
2    :parameters (?c - Convoy  ?r1 - RailCab  ?r2 - RailCab  ?t1 - Track
3            ?t2 - Track  ?t3 - Track)
4    :duration (= ?duration 4)
5    :condition
6      (at start (and
7        (not (= ?r1 ?r2))  (not (= ?t1 ?t2))  (not (= ?t1 ?t3))
8        (not (= ?t2 ?t3))
9        (member_Convoy_RailCab ?c ?r2)
10       (on_RailCab_Track ?r1 ?t1)  (on_RailCab_Track ?r2 ?t3)
11       (next_Track_Track ?t1 ?t2)  (next_Track_Track ?t2 ?t3)
12       ...  % checking for locks
13     ))
14   :effect (and
15     (at start (and
16       ...  % locking
17     ))
18     (at end (and
19       (not (on_RailCab_Track ?r1 ?t1))
20       (member_Convoy_RailCab ?c ?r1)
21       (on_RailCab_Track ?r1 ?t2)
22       ...  % unlocking
23     ))
24   )
25 )
```

Every object in the story pattern – irrespective of whether an unchanged object or an object that is going to be created or deleted – is mapped to a parameter of the action. These parameters are checked for inequality in lines 7 and 8 because we employ injective matching in the graph transformation system.

Required links, i.e. unchanged links and links that are going to be deleted, cause the remaining literals in the condition of Listing 2. If there were forbidden links in the story pattern, they would have been translated into negative literals. The negative literal in the effect (line 19) is caused by the `on` link that is to be deleted and the two positive literals (lines 20 and 21) by the member and on links that are to be created.

The translation of joinConvoy does not cover the case of a forbidden pair. While a forbidden link can simply be mapped into a negative literal, a forbidden pair, e.g. the NAC in the story pattern breakConvoy, has to be mapped into a negative existential quantification over the conjunction of object type and adjacent link that connects the object to the LHS. An isolated forbidden object, i.e. when no connecting object exists, can simply be mapped into a negative existential quantification involving only the active predicate. In both cases, inequality conditions are added if the object type has already been matched as a parameter, to be in accordance with employing injective matching. A more elaborate explanation of the mapping from non-durative graph transformation rules to PDDL action schemata that also covers attribute expressions is given in [TK11].

Locking functionality. Now we turn to the extensions of our translation scheme to integrate the locking mechanism we outlined in Section 3. First, the declaration of predicates and functions has to be extended to include the declaration of locks. Listing 3 shows the generated predicate and function declarations for the locks. For clarity, only declarations for the link between Convoy and RailCab are shown. The first six lines specify the read and write locks for ordinary objects and links. There is one pair of locks (read and write lock) for each object (lines 2 and 5) and one pair of locks for each link in the class diagram (lines 3 and 6). Write locks on objects and links are realized as predicates (exclusive lock, true means locked) because an object or link may not be accessed in any way if it is being deleted at the moment (or created in case of a forbidden link). As long as a reconfiguration does not manipulate an object or a link, a concurrent access is allowed. Therefore, read locks are realized as functions (shared lock, greater than 0 means locked).

Listing 3: Generated declaration of locks

```
1  % declaration of write locks for objects and links (in :predicates)
2    (writeNode_active  ?object - Object)
3    (writeEdge_member_Convoy_RailCab  ?convoy - Convoy  ?railcab - RailCab)
4  % declaration of read locks for objects and links (in :functions)
5    (readNode_active ?object - Object)
6    (readEdge_member_Convoy_RailCab  ?convoy - Convoy  ?railCab - RailCab)
7  % declaration of read locks for forbidden pairs (in :functions)
8    (readAdjacentToSource_member_Convoy_RailCab  ?convoy - Convoy)
9    (readAdjacentToTarget_member_Convoy_RailCab  ?railcab - RailCab)
10 % declaration of write locks for forbidden pairs (in :functions)
11   (writeAdjacentToSource_member_Convoy_RailCab  ?convoy - Convoy)
12   (writeAdjacentToTarget_member_Convoy_RailCab  ?railcab - RailCab)
```

The idea of the remaining locking predicates (lines 7–12) is more subtle. Objects within NACs cannot be locked via any of the aforementioned locking predicates (lines 1–6) because these objects do not exist in the current configuration, i.e. there is no explicit object in PDDL's propositional state that represents the object in the NAC. Instead, locking information is added to the objects they are connected to. This, of course, restricts our approach to specific kinds of NACs, namely forbidden links and forbidden pairs.

Locking of forbidden links (and links that are being created) is already supported via the locking predicates for links. Locking of forbidden pairs is supported by the functions in lines 8 and 9 which – pictorially speaking – add locking information to the connecting object. For each link predicate, there is a pair of locking predicates: the first (second) predicate is used to prevent the creation of a target (source) object that is connected to the source (target) object via a link of the same type and direction than the link within the forbidden pair. Objects that are being added are locked in a similar fashion via the functions in lines 11 and 12. However, for objects that are being added by a rule, locking information is attached to all object that the new objects are going to be connected to, i.e. for every appearing link that is connected to the appearing object a write lock for the pair of object and link is acquired. If there is no such object, then there is no need to lock anything since isolated objects do not interfere with any other object or link. Note, that the write locks are realized as functions instead of predicates because it is possible to apply more than one reconfiguration creating object connected to the same existing object concurrently. Also note, that while we support only two specific kinds of NACs, i.e. forbidden links and forbidden pairs, we support any kind of RHS.

Listing 4: Locking literals of `joinConvoy` to support (required) objects

```
1  % checking for locks (in :condition)
2    (not (writeNode_active ?c))  (not (writeNode_active ?r1))
3    (not (writeNode_active ?r2))  (not (writeNode_active ?t1))
4    (not (writeNode_active ?t2))  (not (writeNode_active ?t3))
5  % locking (in :effect, at start)
6    (increase (readNode_active ?c) 1)  (increase (readNode_active ?r1) 1)
7    (increase (readNode_active ?r2) 1)  (increase (readNode_active ?t1) 1)
8    (increase (readNode_active ?t2) 1)  (increase (readNode_active ?t3) 1)
9  % unlocking (in :effect, at end)
10   (decrease (readNode_active ?c) 1)  (decrease (readNode_active ?r1) 1)
11   (decrease (readNode_active ?r2) 1)  (decrease (readNode_active ?t1) 1)
12   (decrease (readNode_active ?t2) 1)  (decrease (readNode_active ?t3) 1)
```

We will now treat the generation of locking literals for the conditions and effects of action schemata and turn to the example of `joinConvoy` again. Listing 4 shows the locking literals generated to support the locking of required objects. For every positive literal that represents the existence of an object, a negative locking literal is added to the condition of the durative action (lines 2–4). As a result, the action is applicable only if none of the required objects (including objects being deleted) has been locked for writing. A shared read lock is acquired when the action is scheduled to begin (lines 6–8) and released when it ends (lines 10–12). The locking literals of required, i.e. unchanged and deleted, links are realized similarly. The same holds for forbidden links since they are translated into the same, yet negative literals as required links.

Although the story pattern `joinConvoy` does not contain forbidden pairs itself, it generates locking literals for the support of forbidden pairs. This is necessary to avoid conflicts between forbidden pairs and the creation of links: a forbidden pair might interfere with creating a link because the link could be adjacent to an object of the same type as the object in the forbidden pair. To avoid this, further locking literals are generated for every

Listing 5: Locking literals of `joinConvoy` to support forbidden pairs

```
1  % checking for locks (in :condition)
2    (= (readAdjacentToSource_member_Convoy_RailCab ?c) 0)   % link is being created
3    (= (readAdjacentToTarget_member_Convoy_RailCab ?r1) 0)  % link is being created
4    (= (readAdjacentToSource_on_RailCab_Track ?r1) 0)       % link is being created
5    (= (readAdjacentToTarget_on_RailCab_Track ?t2) 0)       % link is being created
6  % locking (in :effect, at start)
7    (increase (writeAdjacentToSource_member_Convoy_RailCab ?c))   % link is b. created
8    (increase (writeAdjacentToTarget_member_Convoy_RailCab ?r1))  % link is b. created
9    (increase (writeAdjacentToSource_on_RailCab_Track ?r1))       % link is b. created
10   (increase (writeAdjacentToTarget_on_RailCab_Track ?t2))       % link is b. created
11 % unlocking (in :effect, at end)
12   ...  % analogue
```

appearing link. Listing 5 shows these locking literals. The locking literals in lines 2–5 state that no forbidden pair lock may be acquired for any of the objects that are adjacent to the appearing links. Such a lock will only be acquired if the story pattern of another reconfiguration that is applied concurrently has a forbidden pair connected to one of these objects. The locking literals in the effect (lines 7–10) itself acquire write locks of forbidden pairs to guarantee that no concurrent reconfiguration with a forbidden pair connected to one of these objects will be applied concurrently. Reconfigurations with forbidden pairs check for these write locks in their condition.

Implementation and application. The proposed translation scheme was implemented for the FUJABA TOOL SUITE using *Xpand*, a code generation language for EMF models. Running the translator on a given model, i.e. a class diagram and a set of story patterns, generates the corresponding PDDL planning domain.

Our application scenario includes reconfigurations to move RailCabs or convoys of RailCabs and reconfigurations related to convoy de-/instantiation and membership change. All these reconfigurations are available in the generated planning domain. The planning problems associated with the generated domain consist of 30 track segments, 2–4 RailCabs, a few junctions and specify initial and target track segments for the RailCabs. Such problems can be solved by SGPlan$_6$ within a few seconds: our planning tasks took approx. 1.4, 3.8, and 10.1 seconds (involving 2, 3, or 4 RailCabs, resp.) on an Intel Core i7-2640M running at 2.8GHz with 8GB RAM. The generated reconfiguration plans take advantage of parallel execution of actions when possible, while guaranteeing that concurrently executed actions do not interfere with each other. With regard to the application scenario, this means that RailCabs operate in parallel if they are sufficiently apart from each other, but wait for the execution of other RailCabs' reconfigurations if necessary, e.g. to clear a common track segment.

Listing 6 shows an excerpt of a resulting plan for the problem instance involving 4 RailCabs. During the interval [60–64], `railcab2` and `railcab3` operate in convoy mode. From 64 to 68, they break up the convoy operation because the underlying domain specifies a Y junction between `track19`, `track20`, and `track25`, and they need to move along different routes to arrive at their target locations. To do so, `railcab2` has to fall back, i.e. it still occupies `track19` at 68. Concurrently, i.e. during the interval [60–68], `railcab0` moves from `track16` to `track18` but waits from 68 to 72 to not crash into `railcab2`.

Listing 6: Excerpt of a resulting plan for 4 RailCabs

```
60.041: (MOVE railcab0 track16 track17) [4.0000]
60.042: (MOVECONVOY convoy0 railcab2 railcab3 track18 track19 track20) [4.0000]
64.043: (BREAKCONVOY convoy0 railcab2 railcab3 track19 track20 track21) [4.0000]
64.044: (MOVE railcab0 track17 track18) [4.0000]
68.045: (MOVE railcab3 track21 track22) [4.0000]
68.046: (CREATECONVOY convoy0 railcab2 railcab1 track19 track25 track26) [4.0000]
72.047: (MOVE railcab3 track22 track23) [4.0000]
72.048: (MOVECONVOY convoy0 railcab2 railcab1 track25 track26 track27) [4.0000]
72.049: (MOVE railcab0 track18 track19) [4.0000]
```

5 Related Work

While planning and scheduling is a discipline in artificial intelligence research that has made many advances in the last decades, only few moves have been made to tie planning techniques with the software engineering domain, e.g. [VSF$^+$09] and [SKM07]. The most promising approaches rely on graph transformation systems as an underlying formalism to specify the planning tasks because of their intuitive representation and close association to model-based software engineering. An early attempt into this direction came from Edelkamp and Rensink [ER07]. They showed manual translations from planning tasks specified with graph transformation rules into PDDL and identified some advantages of planning directly on graph transformation systems: the possibility to reduce the state space by representing isomorphic graphs only once and the support for dynamic object creation and deletion. These advantages gave reason for techniques that directly use graph transformation systems for planning, like [RW10] and [EW11]. In [RW10] the planning task is solved by transforming it into a model checking problem, in [EW11] by using heuristic search techniques. None of these approaches support time-consuming reconfigurations.

Tichy and Klöpper [TK11] were first to present an automatic translation of graph transformation rules into PDDL actions. The support for time-consuming reconfigurations was addressed in terms of stereotypes; concurrency issues were not treated. Meijer [Mei12] also provides a translation between graph transformations and PDDL but does not cover time or durative actions. The developed translator works in both directions, i.e. planning tasks formulated in PDDL can also be translated back into a graph transformation system. As opposed to our technique, the main focus of [Mei12] lies on the backward direction. The employed graph transformation tool, however, needs to support existential quantification on edges to match the semantics of PDDL[2].

[2]In PDDL, a literal that is going to be deleted by an action does not have to be present if it is not required in the precondition. In such a case the action is still applicable but does not change the literal.

6 Conclusion and Future Work

We presented a model-based approach for planning time-consuming architectural reconfigurations for self-adaptive systems. Our technique solves planning tasks for graph transformation systems by translating them into an input for efficient off-the-shelf planning systems. It computes temporal plans where each reconfiguration step has its own associated duration and reconfiguration steps can be carried out in parallel. Unintended interferences, e.g. the deinstantiation and use of a software component at the same time, are avoided thanks to the locking mechanism that we integrated into the translation scheme.

Currently, we investigate the use of domain-independent heuristics for planning techniques that are applied directly on graph transformation systems as an alternative approach. We plan to do a detailed evaluation comparing the efficiency of planning directly on graph transformation systems with the efficiency of running off-the-shelf planning systems on corresponding PDDL models generated by our translator.

As for our modelling approach and its locking mechanism, we plan to extend our approach with *required concurrency*, cf. [CKMW07]: we should be able to specify a reconfiguration that explicitly requires the concurrent application of another reconfiguration. Currently, locks set by a reconfiguration can only *restrict* other reconfigurations from being applicable (which is why we called them locks), instead of *enabling* their applicability (like a window of opportunity for another reconfiguration).

References

[BBB$^+$12] S. Becker, C. Brenner, C. Brink, S. Dziwok, R. Löffler, C. Heinzemann, U. Pohlmann, W. Schäfer, J. Suck, and O. Sudmann. The MechatronicUML Design Method – Process, Syntax, and Semantics. Technical report, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, 2012.

[BBG$^+$06] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *28th Int. Conf. on Software Engineering (ICSE 2006)*. ACM Press, May 2006.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specification. In *ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*, 2004.

[CKMW07] William Cushing, Subbarao Kambhampati, Mausam, and Daniel S. Weld. When is Temporal Planning Really Temporal? In *20th Int. Joint Conf. on Artificial Intelligence (IJCAI 2007)*, pages 1852–1859. Morgan Kaufmann Publishers Inc., 2007.

[CWH06] Yixin Chen, Benjamin W. Wah, and Chih-Wei Hsu. Temporal Planning using Subgoal Partitioning and Resolution in SGPlan. *J. Artif. Intell. Res. (JAIR)*, 26:323–369, 2006.

[EHK$^+$97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.

[ER07] Stefan Edelkamp and Arend Rensink. Graph Transformation and AI Planning. In *Int. Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS 2007)*, September 2007.

[EW11] H.-Christian Estler and Heike Wehrheim. Heuristic Search-Based Planning for Graph Transformation Systems. In *Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2011)*, pages 54–61, 2011.

[FL03] Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research (JAIR)*, 20:61–124, 2003.

[FNTZ98] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language based on the Unified Modeling Language. In *6th Int. Workshop on Theory and Application of Graph Transformations (TAGT 1998)*, 1998.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising Software Architectures for Distributed Systems. In *Workshop on Self-Healing Systems (WOSS 2002)*, pages 33–38, 2002.

[KM07] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE 2007)*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[LM98] Daniel Le Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.

[Mei12] Ronald Meijer. PDDL Planning Problems and GROOVE Graph Transformations: Combining Two Worlds with a Translator. In *17th Twente Student Conference on IT*, June 2012.

[RW10] Malte Röhs and Heike Wehrheim. Sichere Konfigurationsplanung selbst-adaptierender Systeme durch Model Checking. In J. Gausemeier, F. Rammig, W. Schäfer, and A. Trächtler, editors, *Entwurf mechatronischer Systeme*, volume 272 of *HNI-Verlagsschriftenreihe*, pages 253–265. Heinz Nixdorf Institut, 2010.

[SKM07] Ron M. Simpson, Diane E. Kitchin, and T. L. McCluskey. Planning Domain Definition Using GIPO. *Knowledge Engineering Review*, 22(2):117–134, June 2007.

[SWW11] Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. Sound and Complete Abstract Graph Transformation. In *14th Brazilian Symposium on Formal Methods (SBMF 2011)*, pages 92–107, September 2011.

[THHO08] Matthias Tichy, Stefan Henkler, Jörg Holtmann, and Simon Oberthür. Component Story Diagrams: A Transformation Language for Component Structures in Mechatronic Systems. In *4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4)*, pages 27–38, 2008.

[TK11] Matthias Tichy and Benjamin Klöpper. Planning Self-Adaptation with Graph Transformations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Int. Symp. on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2011)*, volume 7233 of *LNCS*. Springer Verlag, 2011.

[VSF$^+$09] Tiago Stegun Vaquero, José Reinalda Silva, Marcer Ferreira, Flavio Tonidandel, and J. Christopher Beck. From Requirements and Analysis to PDDL in itSIMPLE3.0. In *19th Int. Conf. on Automated Planning and Scheduling (ICAPS 2009)*, 2009.

[WF02] Michel Wermelinger and José Luiz Fiadeiro. A Graph Transformation Approach to Software Architecture Reconfiguration. *Science of Computer Programming*, 44(2):133–155, August 2002.

SE 13
SOFTWARE ENGINEERING

Transferberichte

Model-Driven Multi-Platform Development of 3D Applications with Round-Trip Engineering

Bernhard Jung, Matthias Lenk

Virtual Reality and Multimedia
TU Bergakademie Freiberg
Freiberg, Germany
{jung,lenk}@informatik.tu-freiberg.de

Arnd Vitzthum

University of Cooperative Education
Berufsakademie Sachsen
Dresden, Germany
arnd.vitzthum@ba-dresden.de

Abstract: While model-driven approaches are nowadays common-place in the development of many kinds of software, 3D applications are often still developed in an ad-hoc and code-centric manner. This state of affairs is somewhat surprising, as there are obvious benefits to a more structured 3D development process. E.g., model-based techniques could help to ensure the mutual consistency of the code bases produced by the heterogeneous development groups, i.e. 3D designers and programmers. Further, 3D applications are often developed for multiple platforms in different programming environments for which some support for synchronization during development iterations is desirable. This paper presents a model-driven approach for the structured development of multi-platform 3D applications based on round-trip engineering. Abstract models of the application are specified in SSIML, a DSL tailored for the development of 3D applications. In a forward phase, consistent 3D scene descriptions and program code are generated from the SSIML model. In a reverse phase, code refinements are abstracted and synchronized to result in an updated SSIML model. And so on in subsequent iterations. In particular, our approach supports the synchronization of multiple target platforms, such as WebGL-enabled web applications with JavaScript and immersive Virtual Reality software using VRML and C++.

1 Introduction

Interactive 3D applications including Virtual Reality (VR) and Augmented Reality (AR) play a central role in many domains, such as product visualization, entertainment, scientific visualization, training and education. So far, however, the application of model-driven development (MDD) approaches and visual modeling languages such as UML is by far not as common as in the development of other kinds of software. Arguably, this lack of acceptance of MDD approaches can be attributed to the specifics of the 3D application development process.

First, 3D development is an interdisciplinary process. Essentially, two groups of developers are involved who use completely different tools and terminologies: 3D content developers and programmers. Misunderstandings between the developer groups can lead to an inconsistent system implementation [VP05]. E. g., if the 3D content developer does not follow the conventions for the naming of 3D objects, the programmer cannot address these objects properly via program code. While MDD approaches may be instrumental in avoiding such misunderstandings, general purpose tools based e. g. on UML may be appropriate for programmers, but are certainly inadequate for the often ”creatively oriented” 3D developers. More promising seem domain-specific languages (DSL) specifically geared towards 3D development.

Second, 3D applications are usually developed in a highly iterative fashion. Although generally desirable for all kinds of software, support for an iterative development process is particularly relevant in the 3D domain. *Round-trip engineering (RTE)* [Ant07, Aßm03, VPDMD05] is a model-driven software development methodology that combines forward engineering (model-to-code transformations) with reverse engineering (code-to-model transformations) and synchronizations between code and model to support iterative development. RTE has proven useful in the development of ”conventional” software as exemplified by several existing integrated tools supporting the *simultaneous* editing and synchronization of UML diagrams and program code. However, due to the concurrent development process in conjunction with the preference for very different modeling/programming tools between 3D designers and programmers (and software designers), the use of a (yet to be developed) integrated tool for the various tasks seems not advisable for 3D development. Instead, an approach is preferable where the distinct developer groups each can employ their tools of choice.

Third, 3D applications are often implemented for multiple platforms, from ordinary PCs, over mobile devices, up to immersive Virtual Reality installations. A possible approach could be the use of cross-platform 3D engines. A disadvantage is that this requires the installation of a 3D engine at the user’s site. In web environments, however, users may not have administrative rights to install the necessary plug-in. In highly specialized environments, such as CAVEs, a suitable version of the 3D engine may not be available. Therefore, 3D applications often need to be developed w.r.t. different programming environments, using different programming languages. During the iterative development cycle, cross-platform synchronization should be supported.

In this paper we describe a round-trip engineering approach for the model-driven, iterative development of multi-platform 3D applications. Section 2 introduces this approach at a conceptual level. In Section 3, a longer example of model-driven, round-trip development for 3D applications is given, with WebGL-enabled web browsers and immersive Virtual Reality as deployment platforms. Section 4 gives an overview of the implementation of our round-trip engineering process. We discuss our approach in Section 5 before we finally conclude in Section 6.

2 3D Development with Round-Trip Engineering

This section gives a conceptual overview of the the proposed round-trip engineering approach for multi-platform 3D development (see Figure 1). The involved developer groups are: software designers, who design an abstract model of the 3D application; 3D design-

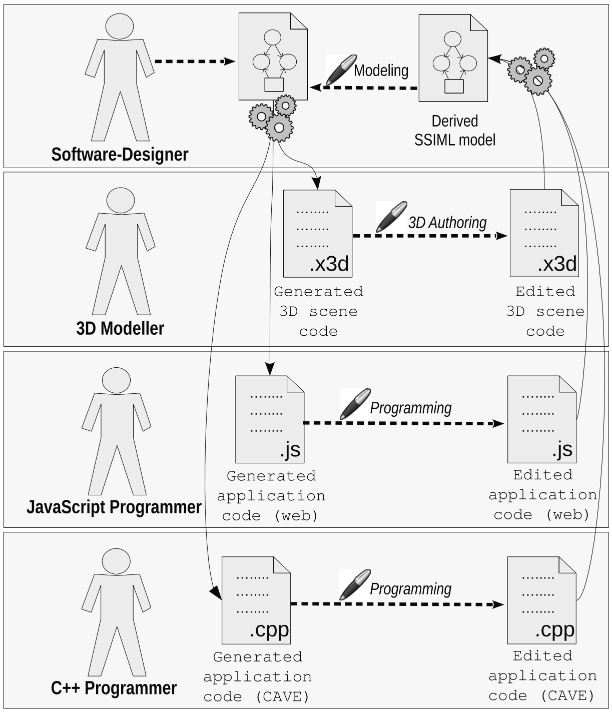


Figure 1: *Model-driven development process with roundtrip engineering, applied to multi-platform 3D application development.*

ers responsible for 3D modeling; and programmers who implement the application logic for the specific target platforms. As example target platforms for the 3D application, we assume WebGL-enabled web browsers with JavaScript as programming language and an immersive Virtual Reality CAVE programmed in C++. Details on the DSL used for application modeling, forward and reverse transformations, as well as other implementation aspects are given in later sections.

Development begins with the specification of an initial abstract model of the 3D application through the software designer. As modeling language, we use the *Scene Structure and Integration Modeling Language* (SSIML) [VP05], a DSL tailored for 3D applications. SSIML models comprise both an abstract description of the 3D scene and a specification of the application logic. Cross-references between the 3D scene description and the application components specify e. g. the events that may be triggered by user interaction with a 3D object and need to be handled by the application logic, or how the application classes modify the 3D scene at runtime. Note that as an iterative development process is assumed, it suffices that the initial SSIML model specifies only a rough first version of the application that can be refined in later iterations.

During the forward engineering phase of the development process, the initial SSIML model is transformed to JavaScript, C++ and X3D code skeletons. The 3D designers' task is to model the individual 3D objects and their composition in a single X3D scene while programmers elaborate the auto-generated JavaScript and C++ code. In the proposed development process, 3D designers and programmers may work concurrently, where both are using different tools appropriate to their task. E. g. 3D designers could model the 3D objects using tools like 3DS Max or Blender, while the programmers use an IDE or a simple code editor.

However, as software development is a non-trivial process, the implementation of 3D and program code may be inconsistent with the current SSIML model in various ways: The implementation could be incomplete or incorrectly reflect the model, e. g. by using different names for 3D objects, application classes or their attributes. Moreover, the implementation may also add parts to the application that may reasonably be reflected in future versions of the SSIML model.

In order to consolidate the various artifacts, the implementation code is then reverse engineered and synchronized with the current SSIML model. For each inconsistency between the SSIML model and the current state of the implementation it has to be decided whether the implementation is incorrect or incomplete or whether the implementation improves the overall application in some way and should therefore be reflected in the next version of the SSIML model. E. g., on the one hand, say, the 3D designer may not care too much about attribute names as only the visual appearance of the 3D objects is usually important for his task. Here, the mismatch should be considered as incorrectness of the implementation. On the other hand, e. g., a programmer may feel that an attribute name in the SSIML model is inconsistent with the conventions of her programming language and uses a different attribute name instead. Here, the changed attribute name may be adopted in the next iteration of the SSIML model. Note that the synchronization step can occur ”on-demand”, at any time of the development. I. e. it is not necessary e. g. that the web application developers wait for the CAVE developers to complete their implementation.

Once the implementation is synchronized with the SSIML model, possibly resulting in updates to the model, the software developer may now further expand the SSIML model. This completes the first cycle of the round-trip engineering process. From the updated SSIML model, again, code is generated in a forward step. Of course, during the forward step it is ensured that the newly generated code still contains the implementation details of the previous iteration, i. e. no code is lost during re-generation. This round-trip process may be repeated for several iterations, until a final version of the 3D application is reached.

3 Example: 3D Development for Web and CAVE

We introduce a small example to better illustrate the proposed 3D development process. The 3D scene is composed of the following 3D objects: A car chassis (Figure 2(a)), separated windows (Figure 2(b)) and different rim types (Figure 2(c)). The application shall provide an interactive part to configure the car, i. e. changing the car’s color, replacing the rims and toggling the visibility of the windows. Interaction is achieved by directly clicking on the respective 3D objects within the 3D scene or by using external GUI elements.

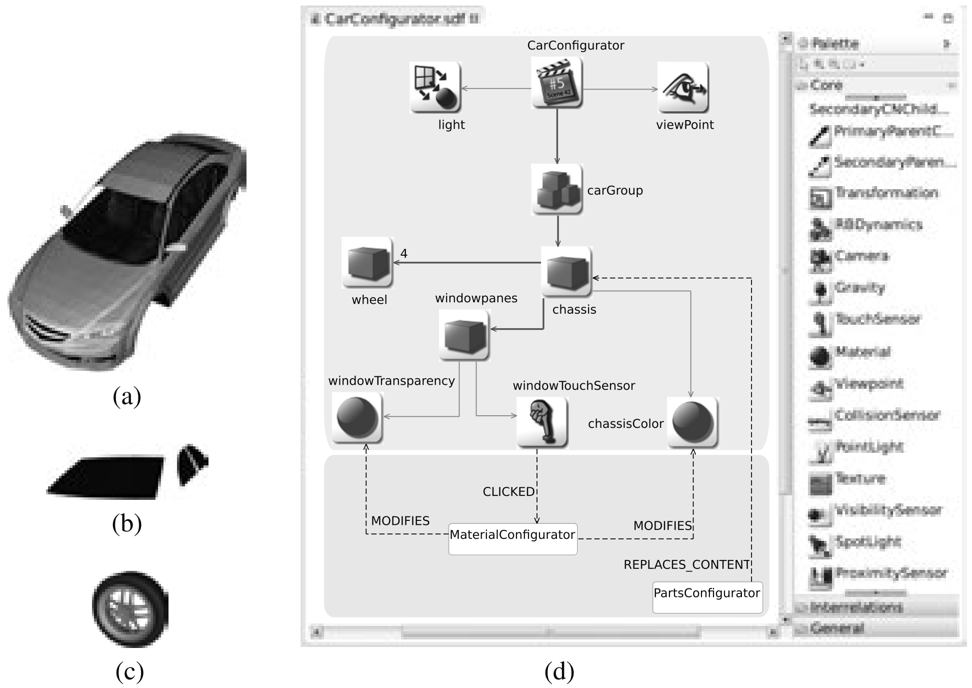


Figure 2: *Left: (a) The stand-alone car chassis with (b) separated windows, (c) one of several wheel types. Right: The SSIML model for the car configuration application in our graphical model editor. Scene model elements are located on the blue (upper) area while elements from the SSIML interrelationship model are positioned on the yellow (lower) area.*

3.1 Modeling 3D applications with SSIML

In the proposed 3D development process, development begins with the specification of an abstract model through software designers. The application model is specified in the DSL SSIML [VP05]. A SSIML model is composed of a *scene model* which allows for modeling abstract 3D scene graphs and an *interrelationship model* to create associations of scene graph elements with application components. The 3D scene graph is composed of parent and child nodes, with some important ones being the *scene root* node, *group* nodes to structure the scene graph, *object* nodes which represent a specific geometric 3D object (e. g., a car chassis) and *attribute* nodes, which can be used e. g. to specify a *transformation* or *material* of an object. Group and object nodes can act as child and parent nodes for other nodes, whereas attribute nodes have no further children. Special attribute nodes, such as *sensor* nodes, are used to trigger events within the 3D scene to notify application components within the interrelationship model. E. g., when a SSIML object node is connected to a touch sensor, this sensor attribute can notify an associated application class about an event, e. g. a mouse click.

```
<!-- id=938c3fb8-b229-40d2-9044-c5a26727bc09 -->
<X3D version="3.0" profile="Interaction">
  <!-- id=a1b5540e-4f9f-4674-8e01-1877a37213b1 -->
  <Scene>
    <!-- id=cb7960db-0141-4f60-ae3d-552ab613decd -->
    <WorldInfo title="CarConfigurator" info="" DEF="..." ></WorldInfo>
    <Transform DEF="generatedTransform1" translation="0 0 0" ...>
      <Group DEF="carGroup">
        <Transform DEF="generatedTransform2" translation="0 0 0" ...>
          <Inline DEF="chassis" url="chassis.x3d"></Inline>
          <Transform DEF="generatedTransform4" ...>
            <Inline DEF="wheel_0" url="wheels.x3d"></Inline>
          </Transform>
          <Transform DEF="generatedTransform5" ...>
            <Inline DEF="wheel_1" USE="wheel_0"></Inline>
          </Transform>
          ...
        </Transform>
      </Group>
    </Transform>
    <DirectionalLight DEF="light" color="1 1 1" ...></DirectionalLight>
    <Viewpoint DEF="viewPoint" position="0 0 10" ...></Viewpoint>
  </Scene>
</X3D>
```

Listing 1: *X3D code skeleton that has been generated from the SSIML model (Figure 2 (d)). Inline nodes are used to include further 3D geometries, e. g. the chassis object. Some IDs and node attributes have been removed.*

The car configuration application can be modelled using SSIML as depicted in Figure 2 (d). Scene model elements which will mainly result in the X3D code are shown on a blue (upper) area, for better illustration. The scene root node has a light and viewpoint attribute and contains the group node "carGroup". This group node contains a "chassis" object which in turn contains four "wheel" objects and the "windowpanes" object. Elements of the SSIML interrelationship model will later be translated to application code and are displayed on a yellow (lower) area. The application class "MaterialConfigurator" is associated with the respective material attributes of the chassis and windowpanes in order to change their color and transparency values. The touch sensor attribute "windowTouchSensor" is connected to the windowpanes object and triggers a "CLICKED" event to the application class when the user clicks on the windowpanes. The second application class "PartsConfigurator" can access the chassis object in order to replace its content i. e. the four wheels with a different kind of wheels.

3.2 Developing the 3D Web Application

Web deployment of the car configuration tool makes use of X3D as scene description language and JavaScript as programming language. The *X3DOM* framework is used for integration of X3D content into HTML pages [BEJZ09]. Modern, WebGL-enabled web browsers can display X3DOM applications natively, without the need for installing a plug-in.

```
function MaterialConfigurator( ) {
  var chassisColor = document
    .getElementById( 'chassis__chassisColor' );
  var windowTransparency = document
    .getElementById( 'windowpanes__windowTransparency' );
  this.windowpanes_CLICKED = function( obj ) {
  };
  //! Insert further application code below !//

  //! Insert further application code above !//
}

function PartsConfigurator( ) {

  var chassis = return document.getElementById( 'chassis' );
  ...
}

function init( ) {
  var materialConfigurator = new MaterialConfigurator( );
  var partsConfigurator = new PartsConfigurator( );
  document.getElementById('windowpanes').addEventListener("click",
      materialConfigurator.windowpanes_CLICKED);
  ...
}
```

Listing 2: *Generated JavaScript code skeleton with functions and member variables to access the 3D scene. Comments, where IDs are stored, have been removed.*

3.2.1 Generation of X3D code from SSIML models

SSIML *scene model* elements (Figure 2, upper blue area) are mapped to X3D nodes in the following way: The SSIML model and the scene root node are transformed to an X3D tag and a scene tag, respectively. Each SSIML group node, e. g. the carGroup, is mapped to a corresponding group tag in X3D. SSIML object nodes are transformed to X3D inline nodes. A generated X3D inline node links to the X3D file specified in the encapsulatedContent attribute of the SSIML object, or, if that attribute is undefined, to a newly generated URL. Additionally, an X3D transform node is generated for these inline nodes in order to support scene composition activities such as translating and rotating the 3D objects to their final positions in the complete 3D scene (Listing 1).

Furthermore, a unique ID – with which each SSIML element is associated – is assigned to each X3D node generated from the SSIML model. This is necessary to track changes made to an element during the edit phase. The ID is bound to its element during the complete development process and may not be modified or removed. Elements without ID will later be treated as newly added elements. Using the X3DOM framework, the generated and refined X3D scene can be included in HTML documents and rendered in modern web browsers (although no functionalities are implemented yet for the application).

3.2.2 Generation of JavaScript code from SSIML models

Similar to X3D code, a corresponding JavaScript code skeleton (Listing 2) is generated to access the 3D scene. The generated JavaScript functions emulate the class-oriented structure of programming languages such as Java or C++, in order to enable consistent multi-

```
class MaterialConfigurator: public ssiml::ApplicationClass {
  public:
    MaterialConfigurator(ssiml::Scene *instance) :
      ApplicationClass(instance),
      chassisColor((SoVRMLMaterial *)scene->getNodeByName("chassisColor")),
      windowTransparency((SoVRMLMaterial *)scene->getNodeByName("..."))
    {}
    bool windowTouchSensor_CLICKED(ssiml::EventHandlerArguments *ea) {
      return false;
    }
  private:
    SoVRMLMaterial *chassisColor;
    SoVRMLMaterial *windowTransparency;
};

// ...

class CarConfigurator : public ssiml::Plugin {
  public:
    CarConfigurator(xs::Host & host, std::string pluginName, ...) :
      Plugin(host, pluginName, sceneFile, enableSmoothing, highlightType) {
        registerApplicationClass<PartsConfigurator>();
        registerApplicationClass<MaterialConfigurator>();

        registerEventHandler<MaterialConfigurator>(ssiml::EventType::CLICKED,
            "windowpanes", &MaterialConfigurator::windowTouchSensor_CLICKED);
    }
  // ...
};
```

Listing 3: *Generated C++ code skeleton with the MaterialConfigurator class and the CarConfigurator scene class. Methods and member variables are generated to access the 3D scene. Comments, where IDs are stored, have been removed.*

platform adaption. JavaScript code fragments are related to the original SSIML model of Figure 2 in the following way: The SSIML application components MaterialConfigurator and PartsConfigurator are translated to JavaScript functions of the same names which create suitable JavaScript objects. The MaterialConfigurator's action relationships to the original SSIML attributes windowTransparency and chassisColor result in member variables of the JavaScript object. These two variables address X3D material nodes which can be used to modify the appearance of the windows and the chassis, respectively. To realize touch sensors with X3DOM, appropriate event listeners (conforming to the HTML event model) are attached to the respective X3D nodes [BEJZ09]. This event handling mechanism is common on many platforms and therefore can easily be adapted. For the windowpanes object, which is connected to a touch sensor in the SSIML model, a corresponding event listener is generated in the init() function. The CLICKED event from the SSIML example is mapped to an HTML click event. Further application logic, e. g. to set the transparency value of the material, will be manually programmed by filling in code stubs or in additional functions. Like generated X3D elements, generated JavaScript elements are also assigned IDs for tracking. The mapping from SSIML elements to corresponding JavaScript elements is more complex than to X3D nodes, since multiple SSIML elements may result in one JavaScript element. E. g., the event handler assignment within the init() function is comprised of the concerned object (i. e. windowpanes), the attached sensor and the relationships which connect the related objects (Listing 2).

3.3 Development of CAVE application

The second platform supported by our round-trip environment is an ultra high-resolution CAVE consisting of 25 projectors (24 full HD plus 1 SXGA+) that offers a much more immersive experience than the web platform. The user can interactively configure the real sized car by using GUI elements on an iPad and by using a flystick for 3D interaction. The 3D scene (Section 3.2.1) is converted from X3D to the VRML format. Functionalities to load, access and display the 3D scene or to realize the event handling mechanism are implemented w.r.t. corresponding libraries of a proprietary CAVE framework. The structure of the generated C++ application (Listing 3) is similar to the JavaScript code (Section 3.2.2). SSIML application classes result in respective C++ classes. Additionally, the class CarConfigurator is generated, in order to initialize the application and to register events. Member variables and stubs for all methods required for the application, such as the "windows_touch" event handler, are also generated in the forward step.

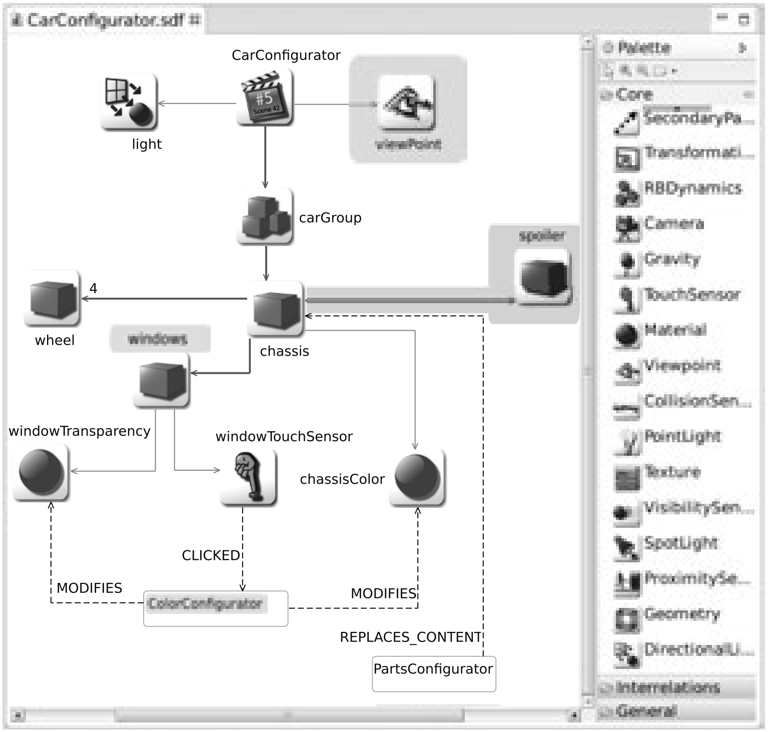
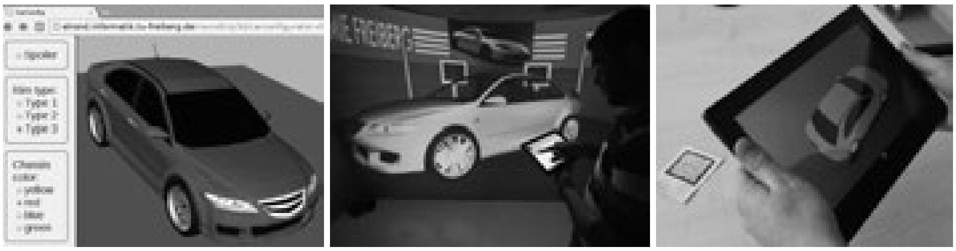


Figure 3: *The second iteration SSIML model for the car configuration application. The "windowpanes" object has been renamed to "windows" and the application class "MaterialConfigurator" to "ColorConfigurator". Furthermore, a "spoiler" object has been added to the new version of the SSIML model.*



(a) (b) (c)

Figure 4: *(a) 3D application in Web Browser; (b) CAVE-version of the application; (c) Prototype of an AR port of the application.*

3.4 Multi-Platform Round-Trip Engineering

During the refinement of the generated code skeletons, model relevant changes to the code can result in an inconsistent 3D application. In order to detect these inconsistencies and to derive an updated SSIML model, the different code bases are synchronized in the reverse phase of the RTE process. For our running example we assume the following three model-relevant changes to the code, which represent the basic operations *update*, *insert* and *delete*: First, 3D designers rename the 3D object “windowpanes” to “windows”. Since JavaScript and C++ programmers rely on the original naming, the windowpanes object is no longer accessible from the program code, which results in an inconsistent application that might crash. Second, 3D designers, having the CAVE application in mind, add a high-detail background to the 3D scene which is however not suitable for the web application due to the smaller graphics processing power in web browsers. And third, 3D designers forget to define the required light source (which could be interpreted, without further information, as intentional deletion). Model relevant changes, of course, may stem from JavaScript programmers, too. For instance, a JavaScript developer renames the class element “MaterialConfigurator” to “ColorConfigurator” since in her opinion only color values are addressed and modified. When transforming the code bases back to the model in the reverse phase, the software designer, ideally in a group decision process with the other developers, may accept or reject such changes. W.r.t. the example, the deletion of the viewpoint (by not defining it) and the addition of the high-detail background are rejected[1] while renaming the windows 3D object and the application class MaterialConfigurator is accepted for inclusion in the next iteration of the SSIML model. Software engineers can now apply further changes to the consolidated SSIML model (Figure 3), e. g. by adding a spoiler object. During the next forward iteration, manually developed source code and changes from the SSIML model will be re-generated to a consistent application. Figure 4 shows the final application, in the web[2] and CAVE variants, as well as a yet experimental AR application on a tablet computer.

[1]CAVE developers may still programmatically add a high-detail background, but the background object will not be part of the platform-independent SSIML model.

[2]Available at: http://elrond.informatik.tu-freiberg.de/roundtrip3d/carconfigurator.xhtml

4 Implementation overview

In this section we briefly describe the implementation of our Eclipse-based Round-trip environment which includes model to code and code to model transformations, model merging and conflict handling. A detailed description of the participating models and transformations is given in [LSVJ12].

4.1 Transformations between model and code

Round-trip engineering combines model-driven forward engineering with reverse engineering as well as a synchronization phase (see Section 4.2). In our implementation, the overall, complex conversion between model and code is realized through a sequence of simpler transformations (see Figure 5). A SSIML model is first converted into its intermediate representation by *model to model* transformations (M2M). The newly created intermediate model (IM) serves as a persistent storage for data and meta data from the complete application, including all model and code artifacts. SSIML to IM transformations (forward phase) and IM to SSIML transformations (reverse phase) are performed with Java using EMF's reflection API [SBPM09]. In the next step *abstract syntax trees* (ASTs) are created from the IM for each platform. We use the hybrid, rule-based transformation language ETL [KPP08] to achieve complex mappings between SSIML elements (in their intermediate representation) and AST elements of the target platform during the forward phase and the reverse phase, respectively. Code serializers and parsers to convert between the ASTs and their concrete syntactic representation are generated through Xtext [EV06] based on a grammar and a meta model of the target platform. Since serializers and parsers only need to recognize model relevant language artifacts, it is not necessary to semantically distinguish between every construct of the target language. Therefore, it suffices that the meta models and grammars cover relevant subsets of the target platform [Jon03]. Due to the similar structure of the generated JavaScript and C++ files, parts of the meta models and IM and ASTs conversions can be re-used.

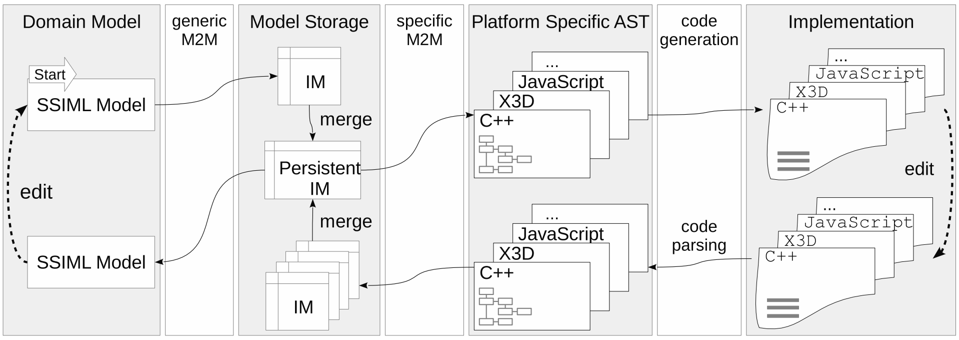


Figure 5: *RTE implementation with participating models and transformations.*

4.2 Synchronization: Model merging and conflict handling

3D modelers and programmers elaborate the generated code bases with different tools appropriate to their task. Therefore, modifications to the code cannot be tracked in real time and thus need to be synchronized with the model in a *non-simultaneous* manner. In order to avoid the need for different merge algorithms for each platform, synchronization is performed on the intermediate layer [LVJ12]. All intermediate models, whether generated from SSIML models in the forward phase or derived from code in the reverse phase, are merged into the persistent intermediate model. Conflicts arise when an element has been modified in different code bases, e. g. an application class has been renamed both in the C++ code and in the JavaScript code. Conflicts and changes to model-relevant elements in the code, such the renaming of the windows object in the X3D code, are automatically detected during merging. Conflict resolution and decision making whether model-relevant changes at the code level should be applied to the next iteration of the SSIML model occurs in an interactive fashion, e.g. by the software engineer in coordination with the different developer teams. When no conflicts are detected during merging, a consistent version of the 3D application has been developed.

5 Discussion

Many existing RTE tools, such as *UML LAB* or *Together*, offer support for software development with focus on *simultaneous* synchronizations of model and code. UML LAB uses templates to synchronize between uniform modeling and general-purpose programming languages, e. g. UML2 class diagrams and Java or C# code. In contrast, the proposed RTE process addresses the special domain of 3D development. 3D and application code is generated from a graphical DSL and elaborated by different developer groups, each using their own specialized development tools. Synchronization of the various artifacts occurs ”on-demand”, in a *non-simultaneous* fashion. The concurrent work style also improves on the typically sequential 3D development process where 3D modeling often strictly precedes the implementation of the application logic.

Our multi-tiered RTE implementation achieves complex model-to-code transformations through a sequence of simpler transformations. In this way, task specific transformation languages can be employed and the process of defining transformations becomes more manageable overall. Furthermore, subdividing the model-to-code transformation chain simplifies the extension to further platforms. Due to the similar structure of the generated code for the different platforms, meta models and IM-to-AST transformations can be reused to some extent. In addition, SSIML-to-IM transformations can be completely reused. A single model-merging algorithm is used instead of *weaving* code of different languages.

Besides SSIML, several other approaches to the structured development of 3D applications have been proposed. The Interaction Techniques Markup Language (InTML) integrates 3D interaction techniques into VR applications, by specifying an abstract XML-based model that can be executed in a runtime environment [FGH02]. However, in this approach the functionality of the program is limited to the scope of its modeling language and multi-platform development depends on the port of the runtime. CONTIGRA [DHM02] aims at high-level and multi-disciplinary development processes of 3D components, that can be transformed to 3D applications. However, a reverse step to achieve iterative development is not possible. APRIL [LS05] can be used to create textual models of AR presentations. These models can be transformed to code for two platforms, while a reverse phase is not possible. SSIML can also be used for modeling AR applications [VH06]. In current work, we are extending our RTE framework to Android-based AR applications (Figure 4(c)).

In the current state of our implementation, a complete round-trip is possible for 3D web applications. I. e. changes to the SSIML model, X3D, and JavaScript code can be fully synchronized with each other and also be applied to the C++ CAVE code. The reverse engineering step for C++ code is currently only partially implemented, such that changes to the C++ code cannot yet be merged into other artifacts. Layout and formatting information of model and source code is not preserved, except in protected regions in JavaScript and C++. Formatting functionalities are provided by programming tools and our model editor.

6 Conclusion

We presented a round-trip engineering approach supporting the multi-platform development of 3D applications. From a common model specified in SSIML, a domain specific language for modeling 3D applications, code skeletons for programs running in modern web browsers and a Virtual Reality CAVE are generated. In a reverse phase, code level changes are synchronized with the SSIML model to support an iterative and concurrent development process. A multi-tiered transformation pipeline is used to implement the RTE process. Splitting up the forward and reverse phases into smaller steps simplifies the extension of our RTE process to further platforms, as only platform-specific transformations need to be implemented for each additional platform. Similarly, as merging occurs between the platform-independent intermediate models, no additional merge algorithm must be implemented when further platforms are considered. The multi-tiered RTE implementation thus scales well with the number of target platforms in 3D application development.

Acknowledgements

This research was supported by the Deutsche Forschungsgemeinschaft (DFG).

References

[Ant07] Michal Antkiewicz. Round-trip engineering using framework-specific modeling languages. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy

L. Steele Jr., editors, *OOPSLA Companion*, pages 927–928. ACM, 2007.

[Aßm03] Uwe Aßmann. Automatic Roundtrip Engineering. *Electr. Notes Theor. Comput. Sci.*, 82(5), 2003. Proceedings of the Fifth Workshop on Quantitative Aspects of Programming Languages (QAPL 2007).

[BEJZ09] Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. X3DOM: a DOM-based HTML5/X3D integration model. In Stephen N. Spencer, Dieter W. Fellner, Johannes Behr, and Krzysztof Walczak, editors, *Web3D*, pages 127–135. ACM, 2009.

[DHM02] Raimund Dachselt, Michael Hinz, and Klaus Meissner. Contigra: an XML-based architecture for component-oriented 3D applications. In *Proceedings of the seventh international conference on 3D Web technology*, Web3D '02, pages 155–163. ACM, 2002.

[EV06] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe 2006*, November 2006.

[FGH02] Pablo Figueroa, Mark Green, and H. James Hoover. InTml: a description language for VR applications. In *Web3D '02: Proceedings of the seventh international conference on 3D Web technology*, pages 53–58, New York, NY, USA, 2002. ACM.

[Jon03] Joel Jones. Abstract Syntax Tree Implementation Idioms. *Pattern Languages of Program Design*, 2003. Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003).

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zrich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.

[LS05] Florian Ledermann and Dieter Schmalstieg. APRIL A High-Level Framework for Creating Augmented Reality Presentations. In *Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, VR '05, pages 187–194. IEEE, 2005.

[LSVJ12] Matthias Lenk, Christian Schlegel, Arnd Vitzthum, and Bernhard Jung. Round-trip Engineering for 3D Applications: Models and Transformations. Preprint 6/2012, Faculty of Mathematics and Informatics, TU Bergakademie Freiberg, 2012.

[LVJ12] Matthias Lenk, Arnd Vitzthum, and Bernhard Jung. Non-Simultaneous Round-Trip Engineering for 3D Applications. In *Proceedings of the 2012 International Conference on Software Engineering Research & Practice, SERP 2012*, 2012.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[VH06] Arnd Vitzthum and Heinrich Hussmann. Modeling Augmented Reality User Interfaces with SSIML/AR. *Journal of Multimedia*, 1(3):13–22, 2006.

[VP05] Arnd Vitzthum and Andreas Pleuß. SSIML: Designing structure and application integration of 3D scenes. In *Proceedings of the tenth international conference on 3D Web technology*, Web3D '05, pages 9–17, New York, NY, USA, 2005. ACM.

[VPDMD05] Ellen Van Paesschen, Wolfgang De Meuter, and Maja D'Hondt. SelfSync: a dynamic round-trip engineering environment. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 146–147, New York, NY, USA, 2005. ACM.

Systematische Rekonfiguration eingebetteter softwarebasierter Fahrzeugsysteme auf Grundlage formalisierbarer Kompatibilitätsdokumentation und merkmalbasierter Komponentenmodellierung

Peter Manhart

RD/EDS
Daimler AG
Wilhelm-Runge-Straße 11
89081 Ulm
Peter.Manhart@daimler.com

Abstract: Mit dem steigenden Wertschöpfungsanteil SW-basierter Funktionen in Fahrzeugen und schnelleren Releasezyklen bei wachsender Variantenkomplexität steigt der Bedarf an praxistauglichen Lösungen zum nachträglichen Austausch minimaler Teilreleases der zugrundeliegenden SW-Komponenten. Wir stellen einen merkmalbasierten Modellierungsansatz für die Komponenten von Steuergerätereleases vor, der auf einer formalisierbaren Kompatibilitäts-dokumentation von Release-Komponenten basiert. Durch zudem klar definierte Abbildungen der Kompatibilitätsaussagen in ein merkmalbasiertes Varianten-modell ermöglicht unser Ansatz eine systematische Rekonfiguration SW-basierter Fahrzeugsysteme.

1. Einleitung

Wettbewerbsrelevante Fahrzeugfunktionen mit hohem Wertschöpfungsanteil, wie zum Beispiel Fahrerassistenzsysteme, sind immer häufiger softwarebasierte Funktionen. Sie werden immer schneller weiterentwickelt und unterschiedlichen Baureihen bereitgestellt. Im Falle von Funktionserweiterungen oder der Anpassung an sich wandelnde gesetzliche Anforderungen stellt sich die Aufgabe der Rekonfiguration: Welche minimale Teilkonfiguration eines Systems muss ausgetauscht werden, um eine Änderung konsistent umzusetzen.

Der hier dargestellte Lösungsansatz entstand im Umfeld der Integration von HW- und SW-Komponenten für ein Kombiinstrument in der Fahrzeugindustrie. In dem Bereich werden nur als wettbewerbsrelevant eingestufte oder stark baureihenspezifische Teilfunktionen intern entwickelt. Die Hardware und die Basissoftware kommen vom Zulieferer. Der Quellcode zugelieferter SW-Komponenten ist in der Regel für die interne Funktionsentwicklung nicht verfügbar.

In der Ausgangssituation waren in dem Entwicklungsbereich folgende, für unsere Betrachtung relevante Entwicklungspraktiken etabliert:

1 Eine Menge systematischer *Anforderungsspezifikationen* für die Komponenten und deren Zusammenspiel.

2 Ein *Versionsmanagement*, das die inkrementelle Weiterentwicklung aller relevanten Entwicklungsartefakte wie Spezifikationen, Lösungen in Form von Modellen, HW-Merkmale und SW-Komponenten, Buildprozessen etc. festhält und dokumentiert.

3 Ein *Releasemanagement* im Kontext des Versionsmanagements, das die Nachverfolgbarkeit von sich zeitlich weiterentwickelnden Releases (Freigabekonfigurationen) sicherstellt und Bausteinversionen zu Konfigurationen gruppiert und diesen einen definierten Reifegrad und einen eindeutigen Bezeichner zuordnet.

Diese Praktiken ermöglichen es bereits (1) herauszufinden, welche Komponentenversionen existieren, (2) zu rekonstruieren, welche Komponentenversionen in welche Releases eingeflossen sind und (3) diese Releases nachzubilden.

Zunächst wurde in dem Bereich merkmalbasiertes Variantenmanagement auf Grundlage der bei uns in der Serieentwicklung bewährten Methode und Werkzeugkette eingeführt. Damit können die Eigenschaften der Komponentenvarianten und ihrer Variationspunkte modelliert und die Konfiguration automatisiert werden. Jedoch wird damit noch nicht die minimale Änderung historischer Konfigurationen unterstützt.

1.1 Rekonfiguration

Wir verstehen unter *Rekonfiguration* den Austausch von Teilreleases in historischen Konfigurationen. Eine Darstellung der Problemstellung in Abgrenzung zum Reengineering findet sich in einer früheren Arbeit [Ma05].

Klassische Konfigurations- und Variantenmanagementansätze, z.B. der CM-Prozess im CMMI-Referenzmodell fordern lediglich die Abbildung von Abhängigkeiten „at given points in time“ [CKS06], also innerhalb einer Konfiguration. Auch der Feature Modeling Process in Kapitel 4.9 von [CE00] beschreibt die Merkmalmodellierung der aktuellen Konfiguration. Für Rekonfiguration müssen jedoch auch Abhängigkeiten zwischen Vorgänger- und Nachfolgerversionen dokumentiert werden. Fehlen diese, wird lediglich der Austausch kompletter Releases durch Nachfolger- / Vorgänger-Releases systematisch unterstützt, aber nicht der Austausch minimaler Teilkonfigurationen. Das ist jedoch selten praktikabel, da beispielsweise aufgrund von HW-Einbauabhängigkeiten oder hoher HW-Tauschkosten der Austausch von HW oft unmöglich oder zu teuer ist. Aber auch der Austausch kompletter SW kann unmöglich oder unwirtschaftlich sein, da die Basis-SW oft nicht extern flashbar verbaut ist oder auch im Falle flashbarer SW ein kompletter Flashvorgang zu lange dauert.

Der hier dargestellte Ansatz zum Kompatibilitätsmanagement zielt darauf ab, dasjenige, möglichst minimale Software-Teilrelease zu ermitteln, das für einen bestimmten Anwendungsfall notwendigerweise ersetzt werden muss. Im folgenden Abschnitt werden unsere Anwendungsfälle und Anforderungen dargestellt, die in unserem Kontext die unmittelbare Anwendung bestehender Rekonfigurationsansätze erschweren.

2. Anwendungsfälle, Anwendungskontext und dessen Bedingungen für Praxistauglichkeit

Unter Abstraktion von Sonderfällen, die mit abgedeckt werden, lassen sich im betrachteten Kontext die folgenden Anwendungsfälle unterscheiden:

2.1 Fall Funktionsänderung: Folge von Spezifikationsänderungen

Bei einer Funktionsänderung wird zunächst die Anforderungsspezifikation einer Komponente weiterentwickelt. Als Folge muss die Umsetzung der Komponente angepasst werden. Dies muss nicht, kann aber in einem größeren oder kleineren Funktionsumfang, sowie in Änderungen der Schnittstelle der Komponente resultieren.

Aus dem Blickwinkel der Rekonfiguration von Komponenten stellt sich die Frage, welche Auswirkungen eine Implementierungsänderung einer Komponente für den Fall hat, dass in einer historischen Konfiguration eine Vorgängerversion der Komponente ersetzt werden muss.

2.2 Fall Fehlerbeseitigung (Bugfix): Austausch fehlerhafter Komponenten

Im Rahmen eines Bugfix bleibt normalerweise die Spezifikation der Komponente stabil, aber die Implementierung muss aufgrund eines von der Spezifikation abweichenden Verhaltens angepasst werden.

Aus Sicht der Rekonfiguration stellt sich die Frage, in welcher Vorgängerversion der Komponente der Fehler zuerst vorhanden war und ob sich diese möglicherweise bereits in der Anwendung befindliche Komponente isoliert austauschen lässt oder ob zusätzliche Maßnahmen nötig sind.

2.3 Anwendungskontext Kombisteuergerät in der Fahrzeugindustrie

Die flashbare und damit prinzipiell austauschbare Software des zugrundeliegenden Kombisteuergerätes besteht aus acht Komponenten. Zwei SW-Komponenten werden intern spezifiziert aber extern entwickelt; der Quellcode ist dem OEM daher nicht verfügbar:

- Die Basisanwendung *APPL* implementiert die Basis-SW incl. Kommunikations-SW, sowie Funktionen zur Ansteuerung von akustischen und optischen Signalen.
- Die Signalschnittstelle *DML* implementiert eine Netzwerkabstraktion, so dass die Funktions-SW netzwerkunabhängig aufgebaut werden kann.

Für sechs Komponenten liegt auch der Quellcode vor, da sie intern entwickelt werden:

- Die Warndatenbank *SMF* enthält länderspezifische Warnmeldungen.
- Die Komponente *GDE* implementiert eine Statechart-basierte Ansteuerung der im Kombiinstrument eingebauten Displaymatrix.
- Die Datenbank *IMG* stellt länderspezifische Symbole bereit.
- Die Datenbank *LNG* stellt länderspezifische Texte bereit.

- Die modellbasiert entwickelte Funktionskomponente *TC* implementiert Fahrerinformationen.
- Die modellbasierte Funktionskomponente *EDF* unterstützt CO_2-sparendes Fahren.

Abhängigkeiten der SW-Komponenten nach außen bestehen beispielsweise zu Varianten der Grafikdisplays oder zu der baureihenspezifischen Signalwelt der Fahrzeugnetzwerke. Inkompatible HW- und SW-Versionen, sowie die Dauer eines kompletten Flashvorganges führen zur Notwendigkeit des Austausches von Teilreleases.

Ein konkretes Beispiel für baureihenbedingte Konfigurationsabhängigkeiten wäre:

- APPL implementiert u.a. die Netzwerkschnittstelle und stellt den anderen Komponenten baureihenspezifische Signale zur Verfügung.
- DML implementiert eine Abstraktion von konkreten, baureihenspezifischen Netzwerkssignalen und stellt diese Signale APPL und anderen SW-Komponenten zur Verfügung. DML hängt damit von APPL ab.
- APPL implementiert auch den Zugriff auf HW wie Leuchten und Zeiger auf Basis der von DML abstrahierten Signale. Damit hängt APPL von DML ab.
- Ein neues Netzwerksignal in APPL würde beispielsweise von DML abstrahiert werden. Eine darauf aufbauende Anzeigefunktion von APPL könnte beispielsweise nicht funktionieren, wenn DML das Signal nicht berücksichtigt.

Die Schnittstelle, die APPL für DML bereitstellt, bezeichnen wir als APPL>DML, die von APPL benötigte entsprechend als DML>APPL.

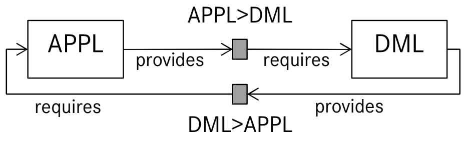


Abbildung 1: Schnittstellen zwischen APPL und DML

Die Schnittstellen fassen wir als *Konfigurationsschnittstellen* auf, sie repräsentieren die äußeren Abhängigkeiten einer Komponente, um diese getrennt vom Funktionsumfang und Semantik ihrer inneren Funktionsimplementierungen zu betrachten.

Der nächste Abschnitt skizziert die in unserem Kontext wesentlichen praktischen Voraussetzungen für Ansätze zum Kompatibilitätsmanagement.

2.4 Bedingungen für Praxistauglichkeit

Über die prinzipielle Lösung der oben genannten Anwendungsfälle hinaus stellt unser Umfeld folgende zusätzliche Anforderungen an eine tragfähige Lösung:

1 Der Austausch einer Teilkonfiguration kann heikel sein, weil bei dem versehentlichen Ersatz einer Komponente durch eine inkompatible Alternative Funktionsstörungen auftreten können. Ein praktikabler Lösungsansatz sollte darum einen zu ersetzenden Teilreleaseumfang hinsichtlich Kompatibilität nachvollziehbar für den Verantwortlichen machen. In unserem Kontext wurde gefordert, die Kompatibilität von Komponentenversionen mit den Eigenschaften der Funktionsänderungen von Komponenten zu begründen.

2 In unseren Fahrzeugsystemen werden viele von Zulieferern entwickelte SW-Komponenten integriert. Da wir keinen Zugriff auf deren Quellcode haben, könnten wir Komponentenabhängigkeiten nur unvollständig aus Quellcodeanalysen ableiten.

3 Die Dokumentation von Komponentenabhängigkeiten ist komplex. Ein praktikabler Lösungsansatz darf darum nicht fordern, globale Aussagen über historische Zusammenhänge zu machen, sondern muss es erlauben, diese zeitnah bei Änderungen im Rahmen der Entwicklung auf Grundlage des momentanen Wissenstandes des Entwicklers zu dokumentieren. Aus der Menge lokal und zeitnah dokumentierter Abhängigkeiten muss sich dann ein global funktionierender Ansatz für das Rekonfigurationsproblem ergeben.

4 Wenige unserer Funktionsentwickler haben Erfahrungen in der Anwendung formaler Softwareentwicklungsmethoden. Ein praktikabler Ansatz muss diesen Funktionsentwicklern ermöglichen, belastbare Ergebnisse mit geringer Fehlerquote zu liefern.

5 Die verfügbare Zeit für Dokumentation ist begrenzt. Der Zusatzaufwand für Kompatibilitätsdokumentation darf deren Nutzen nicht übersteigen.

Diese Bedingungen führen dazu, dass der Schwerpunkt unserer Arbeit nicht der eigentliche Rekonfigurationsalgorithmus ist, sondern die Fragestellung, wie in unserem Umfeld vorab das nötige Abhängigkeitswissen in der Funktionsentwicklung systematisch erfasst und für automatische Rekonfiguration formalisiert werden kann.

Bevor wir unseren Lösungsansatz und dessen Umsetzung beschreiben, werden im nächsten Kapitel die von uns verwendeten Begriffe und eine Kurznotation für diese eingeführt.

3. Begriffe und Notationen

In diesem Kapitel werden die uns wesentlichen Begriffe und Schreibweisen kurz eingeführt, um das Verständnis der im Folgenden verwendeten kompakten Notationen zu erleichtern. Im Anhang werden diese ausführlicher beschrieben.

- Eine *HW- oder SW-Komponente* wird als Zeichenkette notiert, z.B. Signaldatenbank oder S.
- Eine *Version* wird als Zahl geschrieben, z.B. 20120213 oder 0.
- Eine *Variante* wird nicht direkt bezeichnet, sondern im Rahmen der Variantenmodellierung durch Beziehungen zwischen Komponentenversionen abgebildet.
- Eine *Komponentenversion* und deren *Funktionsumfang* wird als versionierte Komponente notiert, z.B. Signaldatenbank120213 oder D0.
- Eine (versionierte) Konfigurationsschnittstelle wird als versionierter Schnittstellenbezeichner der Form SigDB>APPL0 bzw. S>A0.
- Ein Release wird im Rahmen der Variantenmodellierung als Merkmalkonfiguration modelliert.

- Unter *Kompatibilität* verstehen wir die Möglichkeiten und Folgen des Austausches zweier Komponentenversionen in Bezug auf bestehende Releasekonfigurationen. *Abwärtskompatibilität* bedeutet, dass Nachfolgerversionen für Vorgänger eingesetzt werden können, *Aufwärtskompatibilität*, dass Vorgängerversionen für Nachfolger eingesetzt werden können. Kompatibilitätsaussagen haben die Form „–„ für aufwärtskompatibel, „+“ für abwärtskompatibel, „0“ für keine Änderung oder voll kompatibel, „X“ für inkompatibel.
  Aussagen zur Kompatibilität des Funktionsumfangs von Komponentenversionen oder Komponentenversionsschnittstellen können durch Kombination der Notationen ausgedrückt werden, z.B. „D1-“ für eine abwärtskompatible Funktionsänderung von D1 in Bezug auf D0.

4. Lösungsansatz

Der hier vorgestellte Ansatz zielt auf einen Austausch von Komponenten oder Teilreleases mit Kontrolle über Funktionsumfang und unter Wahrung der Schnittstellenintegrität. Der Ansatz basiert auf einer Beschreibung von Komponentenversionen in Releases durch ihre Funktionsumfänge und Schnittstellen, sowie der Beschreibung von Kompatibilitätsbeziehungen zwischen Funktionsumfängen und zwischen Schnittstellen.

Der eigenständige Beitrag unseres Ansatzes ist die formalisierbare Erfassung von Abhängigkeitswissen und dessen schematische Abbildung in ein Merkmalmodell. Die Konzepte, mit denen im Merkmalmodell Kompatibilitätsabhängigkeiten modelliert werden, unterscheidet sich von denen in etablierten Konfiguratoren wie dem Debian Package Manager hauptsächlich dadurch, dass die Abhängigkeitsbeziehungen über Komponentenabhängigkeiten hinaus um Schnittstellenabhängigkeiten ergänzt werden. Diese Abhängigkeitsbeziehungen von Schnittstellen entsprechen denen von Architekturbeschreibungssprachen wie Koala [Om00].

Im ersten Schritt des Ansatzes wird die Releasedokumentation um Kompatibilitätsinformation ergänzt. Diese Zusatzdokumentation ist so aufgebaut, dass im zweiten Schritt eine schematisierte Abbildung in das bei uns für merkmalbasierte Variantenmodellierung [FO90] etablierte Modellierungswerkzeug möglich ist.

4.1 Kompatibilitätsinformation als Erweiterung bestehender Releasedokumentation

Da in unserem Umfeld Teile der Komponenten von externen Zulieferern kommen, können wir nicht wie in Koala Schnittstellenabhängigkeiten automatisch aus Quellcode extrahieren. In unserem Umfeld müssen wir die Abhängigkeitsinformationen vor der späteren Formalisierung zunächst manuell dokumentieren.

In der Ausgangssituation wurden die Releases der Softwarekomponente in einer Tabelle dokumentiert. Eine Zeile der Tabelle enthielt eine Spalte für das Releasedatum, eine für den Releasebezeichner und darauffolgend je eine Spalte für den Versionsbezeichner jeder im Release enthaltenen Komponentenversion. Dies wurde durch ein Bemerkungsfeld abgeschlossen, in dem textuelle Hinweise zu dem Release untergebracht werden konnten.

Diese Tabelle wurde im Rahmen des hier vorgestellten Ansatzes zur Erfassung kompatibilitätsrelevanter Zusatzinformation um folgende weitere Spalten ergänzt:

- Eine Spalte, in der für jede Komponentenversion des Releases beschrieben wird, welche Änderungen an den Funktionen neuer Komponentenversionen vorgenommen wurde. Diese Dokumentation war bis dahin unvollständig.
- Für jede veränderte Komponentenversion eine Spalte zur Dokumentation der Kompatibilität von Funktionsumfang und Schnittstellen.

Beispiel: Die neue Signaldatenbankversion D1 wurde gegenüber D0 um ein Signal erweitert und in der Applikationsversion A1 wurde eine Funktion hinzugefügt, die bei Eintreffen des Signales einen Warnton ausgibt. Der Entwickler würde die neue Situation folgendermaßen dokumentieren:

Applikation	Signaldatenbank D	Änderungsdetails	A	D	…	D>A	..
…	…	…	…	…	…	…	..
A1	D1	D: neues berechnetes Signal Bsm A: neuer Warnton links: BsmWl	+	+		+	..

Die Kompatibilitätsaussagen und ihre Begründungen sind:

- A1 ist abwärtskompatibel zu A0, weil die neue Version alle Funktionen der alten fehlerfrei erfüllt.
- D1 ist abwärtskompatibel zu D0, da D1 alle Signale von D0 liefert.
- Die Ausgangsschnittstelle D>A1 von S1 ist gleichzeitig die Eingangsschnittstelle von A1 und zudem abwärtskompatibel zu D>A0, weil dort alle für den Vorgänger von A nötigen Schnittstellenfunktionen, die Signalisierungen, geliefert werden und weil A alle Signalisierungen des Vorgängers gleich verarbeitet.

Das notwendige Detailwissen für Kompatibilitätsaussagen dieser Qualität ist nur zum Zeitpunkt der Änderung vorhanden. Darum muss die Dokumentation zeitnah zur Änderung erfolgen. Im Sinne optimaler Nutzung von Komponentenvariabilität für die Minimierung von zu ersetzenden Teilreleases ist es sinnvoll, zusätzliche Methoden für die Absicherung von Kompatibilitätsaussagen einzuführen. Beispiele dafür wären eigene Tests auf Kompatibilität oder der Einsatz von geeigneten Analysewerkzeugen.

4.2 Konzept für Kompatibilitätsmodellierung auf Grundlage merkmalbasierter Variantenmodellierung

Die Dokumentation von Kompatibilität, wie sie im letzten Abschnitt dargestellt wurde, eignet sich jedoch in der Form nicht direkt für eine maschinelle Auswertung. Darum sieht der dargestellte Ansatz als nächsten Schritt eine Formalisierung der

Kompatibilitätsinformation als merkmalbasiertes Variantenmodell vor. Im Falle der Rekonfiguration ist ein Modell erforderlich, das die Konfigurationshistorie der Produktevolution inklusive der Kompatibilität der Komponentenversionen abbildet. Dazu müssen die einzelnen Evolutionsschritte unmissverständlich auf Änderungen des Merkmalmodells abgebildet werden. In unserem Modellierungsschema sind neun Hauptanwendungsfälle abstrahiert dargestellt. Die Sonderfälle ergeben sich durch Weglassen der für Kompatibilität eingesetzten Relationen `provides` und `requires` an Merkmalen für Funktionsumfänge und Schnittstellen. Die Abbildung von Kompatibilitätsaussagen in ein Merkmalmodell sieht folgendermaßen aus:

- Komponenten und deren Versionen, Schnittstellen und deren Versionen werden als Merkmale abgebildet.
- Abhängigkeiten zwischen Komponenten und Schnittstellen werden als Beziehungen abgebildet. Beispiele:
  A0 hat Eingangsschnittstelle D>A0: `A0 requires D>A0`
  D0 stellt Ausgangsschnittstelle D>A0 bereit: `D0 provides D>A0`
- Abhängigkeiten zwischen Komponenten und Abhängigkeiten zwischen Schnittstellen werden als Merkmalbeziehungen abgebildet. Beispiele:
  Funktionsumfang von A1 ist abwärtskompatibel zu A0: `A1 requires A0`
  Schnittstelle S>A1 ist abwärtskompatibel zu S>A0: `S>A1 requires S>A0`
  Diese Beziehungen werden bei Aufwärtskompatibilität umgekehrt und bei Inkompatibilität weggelassen.

Wir setzen für die folgenden zwei wichtigen Fälle der Evolution ohne Einschränkung der Allgemeinheit voraus, dass die jeweiligen Komponenten- und Schnittstellen in der Version 0 existieren, also dass beispielsweise A0, D0 oder etwa D>A0 der aktuelle Stand ist.

Fall 1: Ein typischer Anwendungsfall wäre eine gleichzeitige abwärtskompatible Funktionserweiterung einer Applikationskomponente von A0 nach A1. Die folgenden Änderungen bilden den Vorgang im Merkmalmodell ab:

1. Im Applikations-Merkmalteilbaum neue Komponentenversion A1 hinzufügen.
2. Der Komponentenversion A1 die Relation „`requires A0`" hinzufügen.

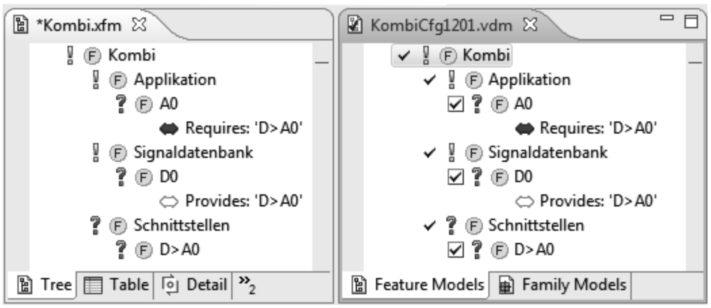
Fall 2: Ein weiterer möglicher Fall wäre eine neue Komponentenversion einer Signaldatenbankkomponente mit abwärtskompatibler Schnittstellenänderung zur Applikation. Die Abbildung eines abwärtskompatiblen Funktionsumfangs würde analog Fall 1 umgesetzt werden. Die Schnittstellenänderung von D>A0 nach D>A1 würde nach folgendem Schema modelliert:

1. Im DML-Merkmalteilbaum neue Komponentenversion D1 hinzufügen.
2. Im Schnittstellen-Merkmalteilbaum die neue Schnittstellenversion D>A1 hinzufügen.
3. Der Komponentenversion D1 die Relation „`provides D>A1`" hinzufügen.
4. Der Schnittstellenversion D>A1 die Relation „`requires D>A0`" hinzufügen.

Mit entsprechenden Schemata ließen sich alle Anwendungsfälle für die Kompatibilitätsmodellierung der Evolution von Funktionsumfang und Schnittstellen abbilden. Die folgenden Abbildungen veranschaulichen den Modellierungsverlauf eines einfachen

Beispiels von Weiterentwicklung. In der Ausgangssituation liegen die Komponenten und ihre Funktionsumfänge in der Version 0 vor (Abbildung 2).

Releases werden als Merkmalkonfiguration abgebildet. Abbildung 2 zeigt auf der rechten Seite die initiale SW-Konfiguration. Verstöße gegen Kompatibilitätsaussagen resultieren in verletzten Beziehungen zwischen den ausgewählten Merkmalen der Konfiguration. Sind alle Beziehungen befriedigt, ist die ausgewählte Konfiguration bei fehlerfreier Modellierung unter Kompatibilitätsgesichtspunkten valide.

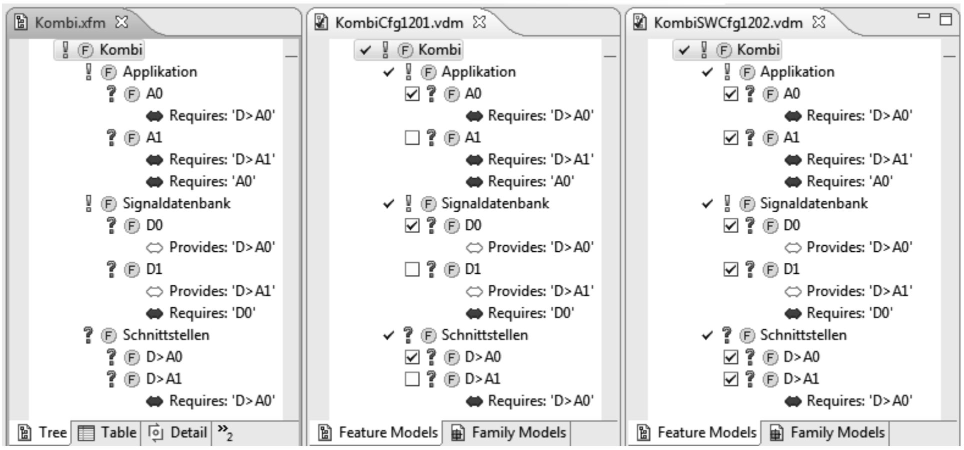


```
*Kombi.xfm
  ! (F) Kombi
    ! (F) Applikation
      ? (F) A0
          Requires: 'D>A0'
    ! (F) Signaldatenbank
      ? (F) D0
          Provides: 'D>A0'
    ? (F) Schnittstellen
      ? (F) D>A0
Tree | Table | Detail | »2

KombiCfg1201.vdm
  ✓ ! (F) Kombi
    ✓ ! (F) Applikation
      ☑ ? (F) A0
          Requires: 'D>A0'
    ✓ ! (F) Signaldatenbank
      ☑ ? (F) D0
          Provides: 'D>A0'
    ✓ ? (F) Schnittstellen
      ☑ ? (F) D>A0
Feature Models | Family Models
```

Abbildung 2: Ausgangssituation Modellierung A und D

Die Modellierung der Kompatibilitätsinformation eines Evolutionsschrittes wie im letzten Abschnitt beschrieben, resultiert in einer neuen Version des Merkmalmodelles. Dieses ist in Abbildung 3 auf der linken Seite dargestellt.



```
Kombi.xfm
  ! (F) Kombi
    ! (F) Applikation
      ? (F) A0
          Requires: 'D>A0'
      ? (F) A1
          Requires: 'D>A1'
          Requires: 'A0'
    ! (F) Signaldatenbank
      ? (F) D0
          Provides: 'D>A0'
      ? (F) D1
          Provides: 'D>A1'
          Requires: 'D0'
    ? (F) Schnittstellen
      ? (F) D>A0
      ? (F) D>A1
          Requires: 'D>A0'
Tree | Table | Detail | »2

KombiCfg1201.vdm
  ✓ ! (F) Kombi
    ✓ ! (F) Applikation
      ☑ ? (F) A0
          Requires: 'D>A0'
      ☐ ? (F) A1
          Requires: 'D>A1'
          Requires: 'A0'
    ✓ ! (F) Signaldatenbank
      ☑ ? (F) D0
          Provides: 'D>A0'
      ☐ ? (F) D1
          Provides: 'D>A1'
          Requires: 'D0'
    ✓ ? (F) Schnittstellen
      ☑ ? (F) D>A0
      ☐ ? (F) D>A1
          Requires: 'D>A0'
Feature Models | Family Models

KombiSWCfg1202.vdm
  ✓ ! (F) Kombi
    ✓ ! (F) Applikation
      ☑ ? (F) A0
          Requires: 'D>A0'
      ☑ ? (F) A1
          Requires: 'D>A1'
          Requires: 'A0'
    ✓ ! (F) Signaldatenbank
      ☑ ? (F) D0
          Provides: 'D>A0'
      ☑ ? (F) D1
          Provides: 'D>A1'
          Requires: 'D0'
    ✓ ? (F) Schnittstellen
      ☑ ? (F) D>A0
      ☑ ? (F) D>A1
          Requires: 'D>A0'
Feature Models | Family Models
```

Abbildung 3: Modellierung eines Evolutionsschrittes und einer Nachfolgekonfiguration

Wie man in den Variantenmodellen in Abbildung 3 sieht, verhält sich die Modellierung, wie man es von der Kompatibilitätssituation erwarten würde. Die bisher bestehende Konfiguration `KombiSWCfg1201` ist nach wie vor gültig, die neu mögliche Konfiguration `KombiSWCfg1202` aus `A1` und `D1` ist zusätzlich gültig.

Für diese Ergebnisse hätte eine der bei uns etablierten Variantenmodellierungstechniken ausgereicht. Nicht aber für die Frage, welche der SW-Komponenten in der jetzt historischen Konfiguration `KombiSWCfg1201` ersetzt werden darf. Ein Austausch von `D0` durch `D1`, der zu `KombiSWCfg1201_UpgradeD` führt, ist kompatibel. Nicht jedoch der isolierte Austausch von `A0` gegen `A1`; in diesem Fall meldet der Variantenmodellierer in `KombiSWCfg1201_UpgradeA` das Fehlen der Schnittstelle `D>A1` (siehe Abbildung 4).

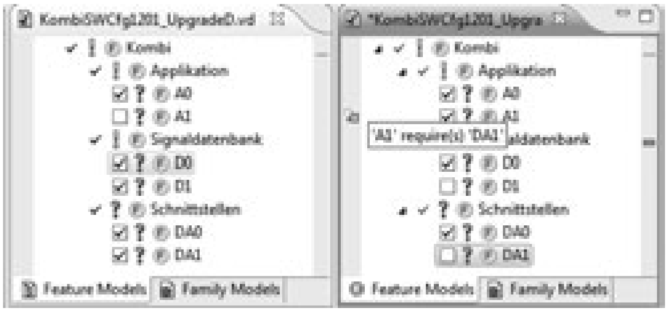


Abbildung 4: Upgradekonfigurationen

5. Prototypische Evaluierung des Konzeptes

Der hier vorgestellte Ansatz wurde für die Software-Rekonfiguration eines Kombiinstrumentes prototypisch umgesetzt. Dieser Anwendungskontext wurde bereits in Kapitel 2 beschrieben.

Während der 12-monatigen Erprobung entstand ein Merkmalmodell für etwa 4 Hauptreleases mit jeweils zwei bis fünf Zwischenreleases. In jedem Release wurden etwa fünf Funktionsänderungen vorgenommen, dokumentiert und modelliert. Dadurch entstand ein merkmalbasiertes Kompatibilitätsmodell mit 62 Merkmalen und 95 Relationen.

Der Dokumentations- und Modellierungsaufwand für die Evaluierung beschränkte sich auf wenige Stunden je Release und wurde als handhabbar akzeptiert. Darum wurde beschlossen, den Ansatz in der Serienentwicklung zu pilotieren und dann im Entwicklungsbereich als Standard zu etablieren.

6. Bestehende Ansätze, Implementierungen und deren Beziehung zu unserem Ansatz

Der Kern unserer Lösung besteht aus drei Bausteinen: Einer Methode zur formalisierbaren Kompatibilitätsdokumentation, einer Struktur zur merkmalbasierten Modellierung von Kompatibilität, sowie einer Systematik der Umsetzung der Dokumentation in die Modellierung. Dabei werden neben den skizzierten Anwendungsfällen alle Use-Cases unter den Bedingungen der Praxistauglichkeit umgesetzt. Im Folgenden werden aus unserer Sicht alternative Herangehensweisen und Arbeiten im Umfeld unseres Ansatzes dargestellt und abgegrenzt.

6.1 Pragmatische Ansätze

Die in der Praxis wahrscheinlich am weitesten verbreitete Herangehensweise ist die textuelle Dokumentation in Texten oder Tabellen. Dabei sichern die Entwickler im Laufe der Entwicklung kompatibilitätsrelevantes Wissen für die spätere Unterscheidung von Komponentenvarianten zur Konfiguration von Systemen. Dieses Wissen enthält auch Anteile, die hilfreich bei der Rekonfiguration historischer Konfigurationen sein können. Da jedoch dieser Vorgehensweise kein formales Modell der Rekonfiguration zugrunde liegt, ist dieses Wissen für einen Fall, den man nicht vorhergesehen hat, regelmäßig unvollständig, so dass nachermittelt oder getestet werden muss. Darüber hinaus sind pragmatische Ansätze sehr stark von individueller Terminologie und individuellem Hintergrundwissen geprägt, so dass die Notizen selbst für Mitglieder des gleichen Teams oder Einsteigern nicht mehr ohne weiteres verständlich sein können. Eine maschinelle Auswertung dieser Informationen ist nicht umsetzbar.

Im Gegensatz dazu liefert der hier vorgestellte Ansatz einen klaren und maschinell auswertbaren Rahmen für die Rekonfiguration. Der Entwickler weiß im Detail, welche Aussagen er im Falle einer neuen Komponentenversion dokumentieren muss und der Modellierer, wie er diese Information in ein maschinelles Rekonfigurationsmodell überführen kann.

6.2 Wissensbasierte Systeme

Es existiert eine Vielzahl von Arbeiten mit dem Ziel der wissensbasierten Konfiguration komplexer strukturierter Produkte. Als einer der ersten praktizierten Ansätze für wissensbasiertes Konfigurieren gilt der regelbasierte Konfigurator XCON, der 1978 für die Konfiguration von DEC-Computern aufgebaut wurde. John stellt in seiner Dissertation [Jo02] einen Ansatz auf Grundlage Constraint-basierter Modellierung über endlichen Domänen vor, der neben Konfigurations-, auch Rekonfigurationsaufgaben adressiert. Mannistö et al. beschreiben [Ma99] eine Methode zur Rekonfiguration, die auf expliziten Rekonfigurationsoperationen und -invarianten beruht. Diese können zum Beispiel aus vergangenen Rekonfigurationen durch fallbasiertes Schließen abgeleitet werden, womit der Vorteil einer gewissermaßen empirischen Absicherung gegeben wäre. Das Problem der technischen Begründbarkeit der Gültigkeit wird allerdings nicht explizit abgedeckt.

Wir adressieren in unserem Ansatz in Ergänzung zu dieser Arbeit die aus unserer Sicht wichtige Problemstellung der Erhebung und Formalisierung von Abhängigkeitswissen.

6.3 LINUX Kernel Configurator LKC und Debian Package Manager

Das LINUX-System enthält einen Kernel-Rekonfigurator für seine Software-Bausteine, der auf einer expliziten Modellierung der Abhängigkeiten zwischen den Konfigurationsoptionen in der KConfig Abhängigkeitssprache basiert. Der Ansatz bewährt sich über eine Periode, in der sich die Anzahl der Merkmale mehr als verdoppelt hat. Die Praxis zeigt jedoch, dass es im Falle neuer Komponentenversionen regelmäßig zu Konfigurationsproblemen kommt, weil nicht alle oder falsche Abhängigkeiten modelliert wurden oder weil die Abhängigkeiten nicht mehr nachvollziehbar sind. Das liegt daran, dass die Abhängigkeiten nicht formal abgeleitet, sondern aus der

Entwicklungserfahrung direkt umgesetzt werden. Eine wesentliche Erkenntnis aus einer Analyse des LINUX-Konfigurators [Lo10] ist ein Mangel an Unterstützung hinsichtlich Nachvollziehbarkeit und Wartbarkeit der resultierenden Repräsentation. Die Autoren extrahierten aus den Commit-Logs auf Seite 12 ihrer Arbeit Aussagen wie ”After carefully examining the code...”, “As far as I can tell, selecting ... is redundant”, “we do a select of SPARSEMEM_VMEMMAP ... because ... without SPARSEMEM_VMEMMAP gives us a hell of broken dependencies that I don’t want to fix ” und “its a nightmare working out why CONFIG_PM keeps getting set” “(emphasis added)”. Bei einer großen Gemeinschaft der Nutzer und falls keine erheblichen Schäden entstehen können ist dies möglicherweise nicht weiter tragisch, weil Fehler schnell auffallen und korrigiert werden. In einem kleinen Entwicklerteam und bei geringer Nutzungsrate oder im Falle kritischer Systemfunktionen ist eine Reifungsphase jedoch kritisch. Im Debian Package Manager werden Paketabhängigkeiten direkt zwischen Komponentenversionen mit den hier im engeren Sinne relevanten Relationen wie *suggests*, *replaces*, *conflicts* und *provides* abgebildet. Eine Herausforderung ist hier wie im LKC die vollständige und konsistente Beschreibung der Abhängigkeiten zwischen den Paketen.

Im Gegensatz dazu leiten sich die Abhängigkeiten in dem hier dargestellten Ansatz aus feineren Aussagen über Unterkomponenten und deren schematischen Umsetzung ab. Derzeit wird zwar die Modellierung noch manuell vorgenommen. Jedoch planen wir die Umsetzung von Kompatibilitätsaussagen zu automatisieren, um das Risiko von Fehlmodellierungen zu verringern und den Modellierungsaufwand zu minimieren.

6.4 Dokumentationsmethodiken

Die Herausforderung der Rekonfiguration eingebetteter Systeme stellt sich spätestens, wenn in der Werkstatt Erweiterungen, Funktionsoptimierungen oder Fehlerbehebungsmaßnahmen umgesetzt werden müssen. In der Dissertation von Köhler [Kö11] wird eine Erweiterung etablierter stücklistenorientierter Dokumentationsmethodiken beschrieben. Dieser Ansatz leitet aus einer Modellierung von Austauschketten für SW-Flashdateien und flashbaren HW-Komponenten unter den Rahmenbedingungen von Rekonfigurationsinvarianten ab, auf welche Weise eine bestimmte Menge von Rekonfigurationszielen erreicht werden kann. Eine Herausforderung dabei ist die Ermittlung funktionierender Austauschketten. Diese werden in dem Ansatz aus freigegebenen Nachfolgekonfigurationen oder auf Grundlage expliziter Tests ermittelt.

Der hier vorgestellte Ansatz basiert im Gegensatz dazu auf einer expliziten Modellierung von Funktionsumfängen und Komponentenschnittstellen von SW-Bausteinen der flashbaren Software zur Zeit der Entstehung der Variabilität. Dadurch werden die technischen Grundlagen der Austauschbarkeit abgebildet und nicht nur die Austauschbarkeit dokumentiert. Zweitens erfolgt die Dokumentation zeitnah und nicht erst vergleichsweise spät im Entwicklungs- oder sogar Dokumentationsprozess. Der dritte Unterschied ist die feinere Auflösung der Modellierung auf die SW-Komponenten, aus denen später die Flash-SW zusammengebunden wird.

6.5 Formale Methoden

Es existieren eine Reihe von Theorien für formales Software Engineering [ET01], die bis auf wenige Ausnahmen noch nicht in unserer Serienentwicklung angekommen sind. Wir verwenden beispielsweise derzeit statische Analysewerkzeuge zum Auffinden von Programmfehlern, aber nicht zur Unterstützung von Konfigurationsaufgaben. In [Tr07] wird beschrieben, dass SAT Solver und Logical Truth Maintenance Ansätze zu Analyse von Featureinteraktionen eingesetzt wurden. Dies würde für den Fall, dass der Quellcode vorliegt, das wichtige Teilproblem der Abhängigkeitsanalyse lösen. Eine weitere relevante Arbeit ist die Beschreibung einer expliziten, formalen Back-Box Spezifikation von Funktionalität auf Grundlage von Timed-Streams im Kontext eines service-orientierten Ansatzes zu Spezifikation von Softwareproduktlinien [HH07]. Der Ansatz sollte prinzipiell die formale Absicherung von Aussagen über Komponentenabhängigkeiten ermöglichen. Wir sehen jedoch vor dem Hintergrund vorhandenen Kenntnisse, Erfahrung, zeitlichen Rahmenbedingungen und der Problemgröße eine große Herausforderung darin, theoretisch fundierte Ansätze dieser Art in unseren Entwicklungskontexten einzuführen.

6.6 Architekturbeschreibungssprachen

Architekturbeschreibungssprachen (ADL) stellen Lösungsansätze bereit, um Komponentenarchitekturen und Komponentenschnittstellen zu modellieren. Beispielsweise ist in [Om00] beschrieben, wie für die ADL Koala aus Quellcodeanalysen Interfacebeschreibungen generiert werden können, wobei auch die Evolution von Systemen adressiert wird. Wir untersuchen derzeit in einer Studie, welche der existierenden Sprachen in unserem Umfeld welchen zusätzlichen Nutzen stiften könnten.

7. Zusammenfassung und zukünftige Arbeiten

Wir haben in diesem Papier einen neuen und praxistauglichen Ansatz zur Erfassung und formalisierbaren Dokumentation von Abhängigkeitswissens für die Rekonfiguration eingebetteter Systeme vorgestellt und eine systematische Abbildung in eine Implementierung auf Grundlage merkmalbasierter Variantenmodellierung dargestellt. Der Ansatz hat sich in der Praxis bewährt und wird daher in der Serienentwicklung eingesetzt. Er basiert auf einer bei uns erprobten Methodik und Werkzeugkette für merkmalbasiertes Variantenmanagement. Mit dem Ansatz sind wir der Lage, minimale Teilreleases zur Lösung einer Rekonfigurationsaufgabe zu bestimmen.

Eine kurzfristig lösbare und bereits in der Weiterentwicklung befindliche Verbesserung betrifft die Automatisierung der schematischen, aber dennoch teilweise umfänglichen, und fehlerträchtigen Änderungssequenzen in der Merkmalmodellierung, die sich regelmäßig im Laufe der Systemevolution als Folge neuer Komponentenversionen ergeben. Hier konzipieren und implementieren wir eine Schnittstelle vom Versionsmanagement in die Modellierung, die auf Grundlage der Kompatibilitätsinformation darauf abzielt, die zugehörigen Änderungen automatisch und fehlerfrei durchzuführen.
Ein mögliches Weiterentwicklungspotenzial des Ansatzes ergibt sich aus der heute schwachen systematischen Absicherung von Aussagen über Kompatibilität: Aufgrund

welcher Kriterien entsteht eine Aussage über die Austauschbarkeit von Komponenten? Derzeit auf Basis von Überlegungen zum Funktionsumfang und den Konfigurations-schnittstellen von Komponenten und deren formalisierbarer Dokumentation. Wir haben begonnen, zur Systematisierung der Dokumentation komponentenspezifische Kriterien-kataloge zu erstellen, um die Vollständigkeit und Widerspruchsfreiheit von Kompatibilitätsaussagen zu erhöhen. Erstrebenswert wäre hier darüber hinaus eine Erhärtung dieser Aussagen durch Kompatibilitätstests und durch analytische Verfahren, wie der statischen SW-Analyse oder aufgrund der automatischen Extraktion von Abhängigkeitsinformation. Ideal wären verifizierte Kompatibilitätsbeziehungen. Wir schätzen es allerdings als große Herausforderung ein, diese Verfahren in unserem Serienprozess umzusetzen.

Nicht zuletzt ist für uns als Forschungs- und Vorentwicklungsbereich auch interessant, wie sich der Ansatz in anderen Entwicklungsbereichen bewährt und wie er über die Zeit und hinsichtlich Größe skaliert. Entsprechende Pilotierung sind nach Serienübergabe in unserem jetzigen Anwendungsbereich geplant.

Literaturverzeichnis

[CKS06] Mary Beth Crissis, Mike Konrad, Sandy Shrum, CMMI: Guidelines for process integration and product improvement SEI series 2006

[CE00] K. Czarnecki, U. Eisenecker, Generative Programming, Addison-Wesley 2000

[FO90] Kang et al, Feature oriented domain analysis (FODA), Technical Report CMU/SEI-90-TR-21, CMU/SEI, 1990

[HH07] Harhurin A., Hartmann J.: A Formal Approach to Specifying the Functionalities of Software System Families, TUM Report, München 2007

[Jo11] John, U..: Konfiguration und Rekonfiguration mittels Constraint-basierter Modellierung Dissertation zur künstlichen Intelligenz, Aka 2002

[Kö11] Köhler, M.: Dokumentationsmethodik zur Rekonfiguration von Softwarekomponenten im Automobil-Service“, Dissertation der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen, 2011

[Lo10] Lotufo R., She S., Berger, T., Czarnecki K., Wąsowski, A.: Evolution of the Linux Kernel Variability Model, SPLC 2010

[Ma05] Manhart P., Reconfiguration – A Problem in Search of Solutions, Configuration Workshop at IJCAI'05

[Ma99] Mannistö T., Soininen T., Toohonen J., Sulonen R. Framework and Conceptual Model für Reconfiguration, Proc. Of WS on Configuration at AAAI’99, Orlando, 1999.

[Om00] Ommering et. al. The Koala Component Model for Consumer Electronics Software, IEEE Computer 2000.

[Tr07] Trinidad et. al, Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines, SPLC 2007

Anhang

Begriffe und Notationen

In diesem Abschnitt werden die uns wesentlichen Begriffe ausführlicher beschrieben, als im Papier, um Missverständnissen vorzubeugen. Für die in der Arbeit relevanten Begriffe wird jeweils eine Notation mit Beispielen angegeben, um kompakte Aussagen zu ermöglichen. Da Syntax in gewissem Umfang beliebig ist, haben wir versucht die im Entwicklungsbereich bereits vorhandenen Schreibweisen um die für hier notwendigen Elemente zu erweitern.

Komponente

In der Automobilindustrie wird der Begriff oft für nur für ein Steuergerät inklusive Software in einem vernetzten Fahrzeugsystem aufgefasst. Hier jedoch wird der Begriff im Sinne der Systemmodellierung hierarchisch und rekursiv verwendet: Komponenten bestehen aus Komponenten, die wiederum aus Komponenten bestehen können und so weiter. Wir unterscheiden HW- und SW-Komponenten.
Generell konfigurationsrelevant bei *HW-Komponenten* sind deren mechanische Eigenschaften wie Bauraum, deren Ein- und Ausgangsschnittstellen, die Netzwerke, die Platinen inklusive deren Speicherausbau (ROM, RAM, NVRAM), Prozessoren und deren Benutzerschnittstellen wie Tester, Drehsteller, Leuchten, Zeigerwerke und Displays.
Konfigurationsrelevante Arten von *SW-Komponenten* sind Basis-SW-Komponenten, welche Betriebssystemfunktionen und eine statisch gebundene Middleware enthalten und die teilweise nicht einfach nachträglich austauschbar sind, z.B. durch Flashen, und Funktionssoftwarekomponenten, die oft aber nicht immer flashbar sind.
Der Einfachheit halber, jedoch ohne Einschränkung der Allgemeinheit betrachten wir hier den einfachsten Fall einer Komponente, die aus einer Menge atomarer HW- und SW-Komponenten besteht. Wenn die Tatsache betont werden soll, dass es sich um eine nichtatomare Komponente handelt, verwenden wir den Begriff *Zusammenbau*.

Notation: Eine Komponentenbezeichner hat die Form `Komp := [A-Z,a-z]*.`

Beispiele: Signaldatenbank, DML, D

Version

Entwicklungsartefakte werden meist in einem inkrementellen, evolutionären Entwicklungsprozess entwickelt. Dadurch ergeben sich aufeinanderfolgende Ausprägungen. Diese bezeichnen wir als *Versionen*.

Notation: Eine Version hat die Form `Ver := [0-9]*`

Beispiele: 20120213, 12001, 1

Variante

Ausprägungen von Entwicklungsartefakten mit ähnlichen, aber nicht gleichen Eigenschaften, die in einem gemeinsamen Betrachtungskontext nebeneinander existieren, bezeichnen wir als *Varianten*. Mit dem Begriff Betrachtungskontext ist ein den Artefakten gemeinsamer Gesichtspunkt, z.B. Releases oder deren Historie, gemeint, der festlegt, warum die Artefakte gemeinsam betrachtet werden.

Notation: *keine*, Varianten werden im Rahmen der Variantenmodellierung durch Konfigurationen im Variantenmodell abgebildet.

Funktionsumfang einer Komponentenversion, eines Softwarebausteins

Den Funktionsumfang oder die Funktionalität einer Version einer HW- oder SW-Komponente oder eines Zusammenbaus fassen wir als Menge der durch sie realisierten Funktionen auf. Damit man Funktionsgleichheit und Funktionsänderungen genau beschreiben und bei letzteren Erweiterungen von Einschränkungen unterscheiden.

Notation: Eine Komponentenversion und deren Funktionsumfang hat die Form `Kv := KompVer`

Beispiele: Signaldatenbank120213, DML0, D0

Bemerkung: Der hier vorgestellte Ansatz kommt in seiner jetzigen Form ohne eine weitere Verfeinerung der Beschreibung des Funktionsumfanges aus. Darum bezeichnet die Komponentenversion gleichzeitig den Funktionsumfang. Eine Erweiterung um die explizite Modellierung der einzelnen Funktionsversionen und -varianten erscheint reizvoll; derzeit existiert jedoch in unserem Umfeld kein wichtiger Use-Case dafür.

Konfigurationsschnittstellen einer Komponentenversion

Damit eine Version einer vernetzten Komponente ihren Funktionsumfang in ihrem Kontext erbringen kann, müssen ihre *Schnittstellen* aus Konfigurationssicht, das heißt die Menge ihrer funktionalen und nichtfunktionalen Beziehungen zu anderen Komponenten abgedeckt sein. Um das sperrige Wort Konfigurationsschnittstelle abzukürzen, verwenden wir im Folgenden den Begriff Schnittstelle.
Es ist aus Konfigurationssicht sinnvoll, die von ihr benötigten *Eingangsschnittstellen*, von den von ihr bereitgestellten *Ausgangsschnittstellen* zu unterscheiden. Analog zum Verzicht auf eine Zergliederung des Funktionsumfanges kommt der hier vorgestellte Ansatz in seiner jetzigen Form ohne eine weitere Verfeinerung der Schnittstellenbeschreibung etwa in Ports, Datagramme etc. aus.

Notation: Die Schnittstellen zwischen den Komponente A und B haben folgende Form:

Komponentenausgangsschnittstelle von A zu B:	`Ko := A>B`
Komponentenausgangsschnittstellenversion von A zu B:	`Kov := A>BVer`
Komponenteneingangsschnittstelle von B zu A:	`Ko := B>A`
Komponenteneingangsschnittstellenversion von B zu A:	`Kov := B>AVer`

Beispiele: SigDB>APPL, DML>A120213, A>D001

Release

Unter einem Release verstehen wir eine identifizierbare Menge miteinander freigegebener Komponentenversionen. Releases existieren auf unterschiedlichen Integrationsstufen wie Gesamtfahrzeug, Fahrzeugsystem oder Komponente und in unterschiedlichen Reifegraden von frühen Test- und Erprobungsreleases bis hin zu serienreifen Freigaben oder Aftersales-Releases.

Notation `RelVer`

Beispiel: Rel120213, Rel1

Kompatibilität von Komponentenversionen

Mit *Kompatibilität* bezeichnen wir die Möglichkeiten und Folgen des Austausches zweier Komponentenversionen in Bezug auf bestehende Releasekonfigurationen. Um Aussagen zu den Folgen eines Tausches von Komponenten machen zu können, müssen darum deren kompatibilitätsrelevanten Eigenschaften beschrieben werden.

Dabei muss zum einen die *funktionale Kompatibilität* gewährleistet werden: Zwei Komponenten mit gleichem Funktionsumfang und gleichen Schnittstellen sind *austauschbar* oder *kompatibel. Abwärtskompatibilität* bedeutet, dass Nachfolgerversionen für Vorgänger eingesetzt werden können, *Aufwärtskompatibilität*, dass Vorgängerversionen für Nachfolger eingesetzt werden können. *Inkompatibilität* bedeutet, dass nach einem isolierten Austausch unakzeptable Änderungen im Funktionsumfang oder Funktionsstörungen durch nicht zusammenpassende Schnittstellen zu erwarten sind.

Notation: Kompatibilitätsaussagen haben die Form:

aufwärtskompatibel:	`Auf := -`
abwärtskompatibel:	`Ab` := +
keine Änderung oder voll kompatibel:	`Voll` := 0
inkompatibel:	`Ink` := X

Um die Kompatibilität des Funktionsumfangs von Komponentenversionen oder Komponentenversionsschnittstellen zu machen, können die entsprechenden Notationen kombiniert werden, wobei hier der Einfachheit des Sachverhalts entsprechend die Kombinationen zusammengefasst dargestellt werden. Die Aussagen und damit die Notationen beziehen sich implizit auf die Vorgängerversion:

Notation: Kompatibilitätsaussagen mit Bezug haben die Form:
auf-/abwärts/voll/inkompatible Funktionsänderung einer Komponente: `Komp-, Komp+, Komp0, KompX`
Komponentenversionenkompatibilität analog: `KompVer- KompVer+, KompVer0, KompVerX`
Schnittstellenversionskompatibilität analog: `A>BVer+ A>BVer-, A>BVer0, A>BVerX`

Agile Software Engineering Techniques: The Missing Link in Large Scale Lean Product Development

Scheerer, Alexander; Schmidt, Christoph T.; Heinzl, Armin; Hildenbrand, Tobias*; Voelz, Dirk*

Institute for Enterprise Systems
University Mannheim
Schloss
68131 Mannheim, Germany
{scheerer, christoph.schmidt, heinzl}@uni-mannheim.de

*SAP AG
Dietmar-Hopp-Allee 16,
69190 Walldorf, Germany
{tobias.hildenbrand, dirk.voelz}@sap.com

Abstract: Many software development companies have fundamentally changed the way they organize and run their development organizations in the course of the last decade. Lean and agile software development became more and more common. Lean focuses on continuous value generation based on a framework of principles known from manufacturing. But how do software developers actually implement these principles in their daily work? Based on insights from several software development teams at a large-scale enterprise software company in Germany, we show that agile software engineering techniques seamlessly integrate into lean product development principles. This paper shows empirical insights on how to implement these principles in a professional context and every-day work.

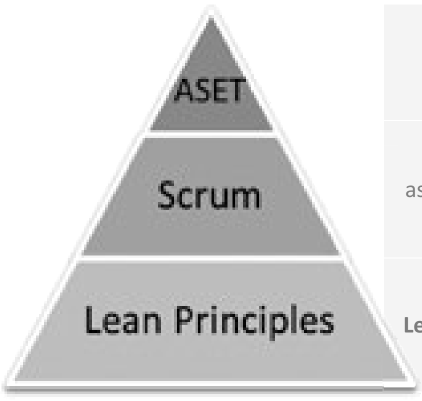
1. Introduction

For many years, large software firms have managed their development organization in a waterfall-like and plan-driven manner to overcome scaling and complexity issues. However, this often resulted in manifold issues such as an ever-growing complexity in the technology stack and an unnecessary level of bureaucratic overhead. Companies realized that a fundamental change was necessary to cope with these scaling problems after it became inevitable that the traditional approach was especially insufficient in large-scale organizations. Lean software development with its underlying principles, which are derived from lean manufacturing [WJ03], promised a solution for many companies. The approach focuses on value and value creation addressing the central identity of the software industry where margins come from innovation and economies of scope, instead of economies of scale [HM12].

The fundamental transition from a traditional plan-driven to a lightweight development approach poses a significant challenge for large companies whose employees have been using the former approach for years. We follow Conboy's [Co09] definitions and consider lean as a framework of principles aiming at the "contribution to perceived customer value through economy, quality, and simplicity". Lean pursues a similar set of objectives as the concept of information systems development (ISD) agility. Agile ISD methods build on “the continual readiness […] to rapidly or inherently create change, proactively or reactively embrace change, […] while contributing to perceived customer value” [Co09].

As Scrum has become a quasi-standard agile method used by many software companies of different sizes, most large-scale lean implementations also started their transition to lean on the team level with this method. However, empirical evidence on both lean and agile methodologies is scarce [PP03], [WCC12], especially when it comes to the adoption by large software vendors. The transition towards lean and agile development remains more a "leap of faith" than a concise engineering effort since too little is known about the idiosyncrasies and pitfalls associated with it.

Are lean principles combined with Scrum really enough to meet the needs of large-scale software development companies? Despite combining lean with agile concepts on the team level by implementing Scrum as process framework, many developers perceive both lean product development and agile software development to be on a yet too general and abstract level for their daily work (see [Re09] and [HM12], for instance). Following the lean idea, its implementation should focus on software products and target those levels where value is created: the development teams, their code, and each individual developer. This is where Agile Software Engineering Techniques (ASET) come into play (cf. Figure 1). We argue in this paper that agile techniques are the missing link to effectively implement and sustain lean product development principles in large scale software companies and finally make them valuable for software developers' daily work.



Specific **agile techniques** [Be01]	Pair programming, test-driven development, continuous integration, refactoring
Agile method as process framework [SB01]	Teams with Scrum master, product owner, developers and an associated manager for people development
Lean principles [PP03]	Eliminate waste, build quality in, learn constantly, deliver fast, engage everyone, keep getting better

Figure 1 Overview of underlying key concepts in our case study

Previous studies already identified agile development techniques as a major success factor in agile software projects, e.g. [Be01]. Hence, we assume a positive outcome of adopting ASET in itself. Based on that, we discuss how agile techniques are adopted in combination with Scrum as process framework and Lean software development with results from an empirical study we carried out at a large enterprise software company in Germany that implements Lean. In order to guarantee the teams exposure to agile techniques we focused on pair programming (PP), test-driven development (TDD), continuous integration (CI), and refactoring (REF) [SS11] which had been introduced to the developers via an intensive team course prior to our investigation.

2. The Research Process

We conducted our study on PP, TDD, CI, and REF during summer and fall of 2011. It comprised twenty one-hour-long interviews with members from four development teams. A questionnaire survey covering the same teams complemented the interviews. All interviewees had attended a one-week training on agile development techniques followed by a three-week coaching phase. To get an in-depth understanding of the techniques and their application in practice, the research team attended the training program as well. This setup ensured that every developer could draw from at least three months of practical and solid theoretical knowledge. The teams had all attended a lean training course within the last two years and were embedded in a lean organization with a focus on continuous value generation. The interviews were transcribed and the data carefully analyzed with statistical and qualitative software packages.

All teams were co-located, had implemented Scrum as a process framework [SS11], [SB01] for at least one year and had several months of ASET experience in a productive setting after the training (cf. Table 1). We interviewed representatives of all Scrum roles (Product owner, Scrum master, and development team member) as well as the teams' line managers and supplemented our analysis with relevant documents concerning the underlying development framework and training materials.

	Team Context		
	Scrum Experience [months]	ASET Experience [months]	Team continuity [months]
Team 1	24	7	24
Team 2	12	3	24
Team 3	48	5	24
Team 4	18	8	12

Table 1 Team contexts

3. The Teams' Attitude towards ASET

The entire training sessions were solely teaching agile techniques with the intention to support software engineers in their daily development work, i.e. having a substantial effect on the development time and gains in the software quality as previously stated by different studies [DB11], [Na08], [MW03], [CW01], [Dy07], [MT04].

Our study results clearly demonstrate that most study participants shared these upfront expectations. A great majority of the respondents generally confirmed (or strongly confirmed) that they enjoyed the taught practices in the first place. If asked whether developers considered ASET beneficial, the overall agreement was even higher than the enjoyment rate (cf. Figure 2). One respondent described the trainings as "finally something tangibly helpful for my daily work after Scrum and lean". In a hypothetical own company, a great majority would adopt these techniques. Overall, no group of strict opponents could be found.

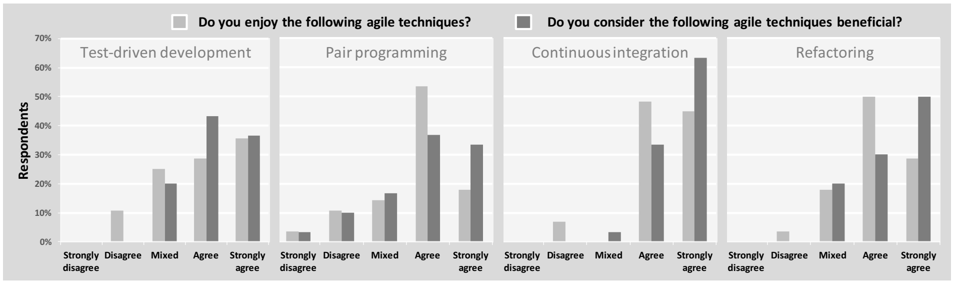


Figure 2 Respondents' enjoyment and conviction of the studied agile techniques

4. Heterogeneous Adoption Patterns

Despite the high level of agreement regarding the benefits and enjoyment of agile techniques, our analysis revealed various unexpected patterns which indicated variations in the degree of the adoption of each agile technique. In particular and contrary to our expectations, our survey results clearly indicate considerable variations among teams intensity adopting pair programming or test-driven development. Table 2 demonstrates these variations with team-based average values for the PP-related question "How much of your development time do you program in a pair?" and "What percentage of code do you write in a test-first manner?" The varying adoption intensity patterns are somewhat surprising, since all teams had participated in the same training program after which they were free to adopt the learned techniques.

	Adoption intensity					
	PP		TDD		REF	CI
	Interview result	*Average development time used for PP*	*Interview result*	*Average of new code written in a test-driven mode*	*Average development time used for REF*	*Average time between code checkins [days]*
Team 1	Low	18%	Low	23%	18%	<2
Team 2	Low	36%	Varying	58%	26%	<1
Team 3	High	60%	High	71%	36%	<1
Team 4	High	76%	High	62%	36%	<2

Table 2 Teams' intensity of adopting PP, TDD, REF, and CI

In-depth insights from our interviews corroborate our survey findings on the heterogeneous adoption patterns among different teams (cf. qualitative indications In Table 2). Moreover, our interview results reveal formerly unknown team-specific factors which help to explain and understand the varying adoption patterns. These factors accentuate respective advantages and disadvantages of the individual techniques and uncover additional contingency factors which hinder or diminish the expected adoption. These will be discussed in the following sections.

4.1. Variations in Pair Programming Adoption Intensity

While programming in a pair, two developers of a team work at one computer, sharing a single keyboard and a mouse for their development work. The partners frequently alternate their active driver and passive navigator roles. Two of the four studied teams applied pair programming to a high degree of their development time with a partner (high adoption intensity), while the remaining teams used the technique occasionally for specific aspects of their work only (low adoption intensity).

According to this pattern, Team 1 (cf. Table 2) is categorized as a team with low adoption intensity. Team members paired if "the work gets more complex or if we see it as helpful to share knowledge". The technique was described as beneficial for security relevant tasks to mitigate the risk of defective code modifications. The pronounced use of a code review system explains the low intensity. Every code change was reviewed in order to achieve a 100% review level. The team regarded code reviewing as superior to pair programming, as knowledge and ideas discussed in a PP session remained mostly within that pair, but not documented. However, pair programming was considered a complementary technique rather than a substitute as the low adoption intensity might suggest. Developers acknowledged their partners' fast feedback and the possibility to jointly solve a coding task, which reportedly increased the team cohesion. The team mentioned several impediments for coding in pairs: potential programming partners from the open-source community worked at different locations and time-zones. Secondly, the team considered the required synchronization within team-internal pairs as a burden for its productivity. This was perceived as particularly unpleasant, since some team members preferred flexible work hours.

For some developers of Team 2, PP was the default mode, but most members only used it parsimoniously (see Table 2). The team attributed the creation of a broader knowledge base among individuals to pair programming which led to higher project flexibility. In particular, the reduction of the "vacation problem" was regarded as valuable, i.e. the team was still productive if individuals were absent. Despite these respected benefits, reported disadvantages out-weighed the advantages for the team in particular situations explaining the variation in this adoption pattern. One developer said he felt stifled in his work flow leading to more stress for him than working alone. Unequivocally, the team shared the opinion to have delivered less features per Scrum sprint.

For Team 3, pair programming was the default work mode. In the planning session, the team decided which tasks to develop in pairs, with coding of new functionality always being developed by two programmers. Still a few disadvantages of pair programming were mentioned as it "can cause conflicts in the team; but the team has to learn how to deal with that". One developer reported to sometimes be "overambitious and to forget to take breaks" which can be more stressful than working alone, since you are busy all the time. Generally, advantages clearly out-weighed the disadvantages for this team. Members emphasized a stimulated discussion culture leading to common quality awareness, and the immediate avoidance of trivial errors as the main benefits.

Team 4's intensity of paired programming was the highest. Almost every coding task was developed by a pair and even non-coding tasks, such as configurations of productive systems, were realized in teams of two: "we try to develop everything possible with a partner". Developers mentioned that they have more fun at work and a better team spirit. The technique was appreciated because of its good fit into the adopted Scrum framework due to its positive effects on teamwork and a common code responsibility. Finally, pair programming reportedly fostered the intra-team discussion culture.

In summary, several impeding factors were discussed which helped to explain low adoption intensity rates. Pair programming is only possible if teams are co-located, if teams share work tasks similar in terms of content, if developers are willing to synchronize their work hours, and for pairing partners who bring the potential and willingness to reach consensus.

Coding with a partner entails three major benefits: a broader common knowledge base leading to increased team flexibility, prevention of trivial errors from the very beginning of coding due to the immediate feedback and increased quality awareness, and finally a closer collaboration among team members fostering team spirit. On the other hand, pair programming was reported to potentially lead to inter-personal conflicts, to be more stressful than working alone, and to interrupt a free flow of thoughts. Some developers reported to perceive a lower development speed of the team due to pairing, e.g. measured as delivered features per Scrum sprint. In general it can be said that through pair programming, the aspect of continuous value generation can be brought to the teams and their daily work.

4.2. Variations in Test-driven Development Adoption Intensity

Test-driven development implies alternate writing of production code and corresponding tests to cover its functionality. In the strict sense, tests are written before the code exists. Consequently, tests fail until the software functionality is implemented consistently. Hence, productive software and its test framework evolve simultaneously.

Among the two low adopting teams of TDD, Team 2's usage can be described as varying due to different subgroups resulting from the team's three technology stacks. In both teams, developers mentioned that "we do not apply TDD consistently; perhaps we use it more in the lower levels of the software stack, parts which have a library character". The TDD process felt unnatural, as stated by one interviewee, "these micro-increments do not come naturally for the seasoned developer. It feels very strange". In general, the consensus on the decisive factor was the focus on tests which resulted in high code coverage.

The benefit of TDD was recognizable "when implementing logic in code which is not straight forward". As Team 1 worked on many user interface development tasks, they described the technique as "not making any sense" since the effort was felt to be disproportionately higher compared to writing a test after the code had stabilized. "Test code can get very complex" as one developer mentioned, "we have had test code which was far more complex than the actual productive code". Nevertheless, in areas where the team decided to use TDD, the increased modularity in the design of the code was evident. Furthermore, the technique increased developers' focus on their current task at hand, as the test suite gave developers more self-confidence and "trust in my own work that lets me sleep well at night". TDD was also viewed as a possible cause of slower development speeds which needed to be factored in at the planning stage. Some programmers felt that "it disturbs the flow of thinking; it is too much back and forth". For others the increasingly small increments between test and productive code writing "are annoying" especially "if I know exactly what needs to be done, then I have to force myself to use TDD".

More senior developers noted the painful transition to TDD, as this programming style required a major change in the thinking process. One impediment in the beginning was the high feature pressure from outside the team which "increased stress on the team members", so that they "couldn't use TDD in a reasonable manner". The technique had been impaired by the large legacy code base the team had to deal with, "it basically has no code coverage and was not written to be easily testable".

In the group of high adopters, TDD was used as "often as possible" with development tasks, although many items were non-coding activities. However, TDD was often not used in the strict sense of minimal steps between writing a test and implementing the productive code, but in a test first fashion. Essentially the steps were enlarged so that fewer context switches between testing and coding were necessary. This can be illustrated by one respondent's answer, "I don't develop tests line by line; that is too strict for me".

A strong advantage of TDD was the confidence brought about by the test-suite. Through this "tightly-knit security net, you can program in a very relaxed manner. One has to keep less open ends in mind". Many interviewees had a high conviction towards agile techniques and their quality of work, with several developers mentioning that "features take more time to implement, but we want to deliver it in high quality". Developers stated that they "can only implement new functionality in existing code, if it is somewhat structured", but this structuredness can only be acquired if they are given "enough time". Moreover, TDD was also regarded as introducing structure into the idea creation process and it was reportedly motivating to develop against the red light of a non-passing test.

One major advantage of test-driven development was recognized in the large test framework produced. This high code coverage let developers "reach easily into existing code and completely refactor certain areas while still being sure that the functionality is implemented correctly". The long term maintainability of the software code was a critical aspect mentioned, "I'm not afraid to touch three year old code, make a change and let the test-suite run to tell me if everything is still working as before".

4.3. Homogeneous Adoption Pattern regarding Continuous Integration and Refactoring

All studied teams continuously integrated new or modified code into their existing code base, as proposed by the literature. CI was described as not helpful on its own, but particularly beneficial in combination with a high test coverage of the code or TDD. One developer said that "continuous integration helps you to permanently get feedback on your code due to the automatic checks against the existing tests every time you integrate [...] we know after each check-in, if the product is shippable, thus we are theoretically ready to release at any time". Another developer expressed his positive attitude as follows: "from continuous integration, I do not see any disadvantages at all". Still another illustrated that "developing software without CI is like flying blind without instruments [...] you just do not know what you are doing, where you are and how much time it will cost to repair something [...] there is no feedback and no transparency". These quotes perfectly represent the teams' unrestrained positive attitude towards the technique.

Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [FB99]. Our survey results reveal that more than 80% of the respondents spent between 20 and 40% of their time refactoring existing code. Nevertheless, members of all teams finished up to 29% of their coding backlog items without refactoring. Team 1, which used the code review tool systematically, reported always paying "attention to make the code design right from the beginning" since messy code would not be accepted in the review anyways.

5. Combining ASET in a Lean and Scrum context to its full benefits

Developers who consider lean principles and Scrum as too abstract find agile techniques to be well suited to transfer and apply the lean concepts into their daily work. Hence, we consider agile techniques as the key to successfully develop software in such a context since they are anything but abstract. We found that ASET were the "missing link" to bring lean to the value stream (cp. also [CC08]).

Building upon a firm base of lean principles, the combination of the studied agile techniques revealed interesting interdependencies between the techniques applied. For instance, developers recognized their PP partner as their personified "bad conscience". Consequently, they felt encouraged to integrate their software more often, to refactor more intensively, and to adhere to the test-driven work mode, even though it was described as time-consuming and costly in the short-term, but admittedly beneficial in the long-run. In particular, combining TDD and CI was considered more advantageous since test-driven development naturally leads to an extensive test-suite making continuous code integrations more attractive.

Our interview results clearly indicate that these agile techniques breathe life into many abstract lean principles and Scrum basics. One developer mentioned that "PP fits perfectly to Scrum and its focus on team empowerment". As discussed before, continuous integration in combination with an extensive test suite enables developers to release their software at any time. Moreover, it increases transparency and allows all team members to be up to date on the current status of their software product. Striking similarities can be seen when compared to [PP03], as both aspects are directly related to the lean principle of "optimizing the whole". Furthermore, developers recognized lean's focus on the "elimination of waste" by adopting continuous refactoring in their daily work. Due to the paired work mode, developers stated to "build quality in" by avoiding trivial errors right from the beginning and to raise a common quality awareness in the team. Moreover, pairing invites developers to "learn constantly" while "engaging everyone" in the team.

Several benefits from adopting agile techniques help development teams to cope with issues prominently found in large scale software development organizations. In such environments, many people work simultaneously on a common code base which can accumulate over several years or decades. Hence, developers face an ever growing technology complexity. The combined adoption of TDD, CI and REF was regarded as one way to tackle this problem; developers attributed a higher modularization of their code to TDD, continuous refactoring helped reinforce coding conventions and thus structuredness and legibility. Consequently, developers can more easily modify the existing code base even if it was previously written by somebody else.

Large scale software development often leads to a certain customer disconnect because of complex organizational setups. TDD and CI introduce fast feedback cycles through a continuous alignment with customer requirements ideally embodied in the test framework. Thus, it fosters a close customer focus which is the prerequisite for delivering value despite an increasing systems complexity.

We conclude that the introduction of agile techniques into a large-scale lean software organization helps developers translate this new mindset into their daily work. For developers, lean and agile is no longer an abstract initiative from management, but a tangible asset which clearly changes the work of every software developer.

6. Recommendations for Large-Scale Lean Implementations

As previous studies have shown, PP, TDD, CI, and REF help developers deliver better software quality. But, there is no free lunch: the improved software quality is clearly traded for a prolonged short-term development time. However, we recommend not to one-sidedly assess gains in software quality in proportion to longer development time, but to equally consider composite effects brought about by applying agile techniques in lean and Scrum contexts. Benefits, such as developers' closer intra-team collaboration, increased common quality awareness or their intensified knowledge transfer, are particularly beneficial in long-term oriented, quality-conscious software development organizations rather than in feature quantity focused ones. For the future, we recommend to balance these trade-offs more consciously according to the requirements of each individual project.

While introducing agile techniques in development teams, you need to take certain contingency factors into account. Developers' personal convictions of the studied techniques predicted their actual adoption the best. Do not expect developers to naturally see long-term benefits, since short-term efforts and change issues can initially seem insurmountable.

If teams can deliberately decide whether and how to adopt agile techniques, as managers among our interviewees recommend, then management should focus on encouraging developers to jointly aspire towards organizational long-term goals, thereby motivating ASET. Training sessions alone are not self-sufficient. Organizations which want to implement lean including ASET need to take the substantial effort for designing technology-specific trainings as well as hands-on coaching into account.

We conclude that sustainable competitive advantage of modern software companies does indeed not only depend on the ability to continuously innovate new products, but also requires the capability to establish organizations and processes to develop, extend and maintain these products at competitive quality, cost and time. Combining lean, Scrum, and ASET helps software companies accomplish exactly that goal.

We are convinced that lean software development in combination with agile techniques offers a major step into the future of software engineering as it enables software companies to adapt quickly to changing environments while focusing on value creation and quality. Future studies should focus around the question of what makes software development teams successful in an agile and lean environment. Furthermore, the investigation of driving factors behind software development process flow across several development teams and organizational levels seem particularly promising. Current trends in the software industry also revolve around the question on how to combine innovative product design and lean development, i.e. how inspiration and innovative ideas for new software products and features happen in an lean and agile development environment (see [HM12] for a first case study).

Bibliography

[Be01] Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 2001.

[CC08] Chow, T.; Cao, D.-B.: A survey study of critical success factors in agile software projects. In: Journal of Systems and Software, Vol. 81, Issue 6, pp. 961-971

[Co09] Conboy, K.: Agility from First Principles: Reconstructing the Concept of Agility in Information Systems Development. In Information Systems Research, 2009, 20; pp. 329–354.

[CW01] Cockburn, A.; Williams, L.: The costs and benefits of pair programming. In Extreme programming examined, 2001; pp. 223–248.

[DB11] Dogša, T.; Batič, D.: The effectiveness of test-driven development: an industrial case study. In Software Quality Journal, 2011; pp. 1–19.

[Dy07] Dybå, T. et al.: Are Two Heads Better than One? On the Effectiveness of Pair Programming. In Software, IEEE, 2007, 24; pp. 12–15.

[FB99] Fowler, M.; Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[HM12] Hildenbrand, T.; Meyer, J.: Intertwining Lean and Design Thinking – Software Product Development from Empathy to Shipment. In: Software for People – Fundamentals, Trends and Best Practices, 2012, pp. 217-237, Springer, Heidelberg

[MT04] Mens, T.; Tourwé, T.: A survey of software refactoring. In Software Engineering, IEEE Transactions on, 2004, 30; pp. 126–139.

[MW03] Maximilien, E.M.; Williams, L. Eds.: Assessing test-driven development at IBM, 2003.

[Na08] Nagappan, N. et al.: Realizing quality improvement through test driven development: results and experiences of four industrial teams. In Empirical Software Engineering, 2008, 13; pp. 289–302.

[PP03] Poppendieck, M.; Poppendieck, T.: Lean software development: an agile toolkit. Addison-Wesley Professional, 2003.

[Re09] Reinertsen, D.: The Principles of Product Development Flow - Second Generation Lean Product Development, 2009, Celeritas Publishing.

[SB01] Schwaber, K.; Beedle, M.: Agile Software Development with Scrum. Prentice Hall, 2001.

[SS11] Schwaber, K.; Sutherland, J.: The Scrum Guide, 2011.

[WCC12] Wang, X.; Conboy, K.; Cawley, O.: "Leagile" software development. An experience report analysis of the application of lean approaches in agile software development. In The Journal of Systems & Software, 2012, 85; pp. 1287–1299.

[WJ03] Womack, J. P.; Jones, D. T.: Lean thinking: banish waste and create wealth in your corporation. Simon and Schuster, 2003.

Modellbasierte Bewertung von Testprozessen nach TPI NEXT® mit Geschäftsprozess-Mustern

Claudia Schumacher, Baris Güldali, Gregor Engels
s-lab – Software Quality Lab / Universität Paderborn
clasch@mail.upb.de, {bguldali, engels}@s-lab.upb.de

Markus Niehammer, Matthias Hamburg
SOGETI Deutschland GmbH 40549 Düsseldorf
{markus.niehammer, matthias.hamburg}@sogeti.de

Abstract: Die Qualität eines zu entwickelnden Softwareprodukts wird entscheidend durch die Qualität des zugehörigen Testprozesses beeinflusst. Das TPI®-Modell ist ein Referenzmodell zur Bewertung der Qualität eines Testprozesses, das mittels Kontrollpunkten den Reifegrad von Testaktivitäten bestimmt. Dabei ist allerdings sowohl die Interpretation des zu bewertenden Testprozesses, welcher in der Praxis häufig gar nicht oder nur informell beschrieben ist, als auch die Interpretation des TPI®-Modells selbst von dem Wissen und den Erfahrungen des bewertenden Experten abhängig. Dies führt unmittelbar zu einer langwierigen, schwierigen und insbesondere subjektiven Bewertung eines Testprozesses. Um eine objektivere, einfachere und effizientere Bewertung zu ermöglichen, wird im Beitrag ein Ansatz vorgestellt, mit dem ein Testprozess als Geschäftsprozess und die Kontrollpunkte des TPI®-Modells in Form von Geschäftsprozess-Mustern mit Hilfe der Modellierungssprache BPMN formal modelliert werden. Auf dieser Basis kann die Qualität eines Testprozesses durch eine systematische Analyse untersucht und bewertet werden. Zur Evaluierung des entwickelten Konzepts wird ein Fallbeispiel eines Testprozesses systematisch bewertet.

1. Herausforderungen bei TPI NEXT®

Die Qualität eines zu entwickelnden Softwareprodukts wird entscheidend durch die Qualität des Testprozesses beeinflusst. Methoden zur Testprozessverbesserung werden bereits seit den 1990er Jahren eingesetzt. Eine der erfolgreichsten dieser Methoden ist TPI® (Test Process Improvement) von Sogeti, dessen aktuelle Version in 2009 unter dem Namen TPI NEXT® [Ew11] veröffentlicht wurde.

Getreu der Aussage von Watts Humphrey „*If you don't know where you are, a map won't help*" [Hu95], enthält TPI NEXT® Kontrollpunkte für verschiedene Test-relevante Kernbereiche (wie z.B. Testmanagement, Testwerkzeuge), um die aktuelle Position eines Testprozesses auf einer Reifegrad-Skala zu bestimmen. Kontrollpunkte sind hierbei Anforderungen an den Prozess, die ähnlich wie bei allen anderen verbreiteten Modellen in natürlicher Sprache formuliert werden. Als Reifegrade verwendet TPI NEXT® die vier Stufen Initial, Kontrolliert, Effizient und Optimierend mit jeweils fester Anzahl an Kontrollpunkten [Ew11].

Derartig natürlichsprachlich formulierte Kontrollpunkte haben ihre Vorteile. Sie sind unabhängig von der Art der Prozessdokumentation, die selbst natürlichsprachlich oder in einer der vielen verschiedenen Modellierungssprachen verfasst sein kann. Darüber hinaus empfinden es erfahrene Prüfer häufig auch einfacher, anhand solcher natürlichsprachlich formulierter Anforderungen einen undokumentierten, gelebten Prozess zu bewerten, z.B. durch Interviews oder Einsichtnahme in Unterlagen.

Die Verwendung der natürlichen Sprache zur Beschreibung von Testprozess und Kontrollpunkten hat aber auch wesentliche Nachteile. Zum einen ist es für Prüfer oft aufwändig und langwierig, abhängig von Qualität, Quantität, Wahrheitsgehalt, Konsistenz, Relevanz und Struktur der gesammelten Informationen sowie der Komplexität des Testprozesses, ein angemessenes Verständnis des Testprozesses zu erlangen. Für die Bewertung eines Testprozesses ist darüber hinaus die Subjektivität der Prüfer ein weiterer Nachteil. Die Bewertung auf Basis natürlichsprachlich formulierter Kontrollpunkte ist stark abhängig von Vorwissen und den individuellen Erfahrungen eines Prüfers.

Um diese Nachteile eines subjektiven, erfahrungsbasierten Bewertungsverfahrens zu beseitigen, haben das s-lab – Software Quality Lab der Universität Paderborn und Sogeti einen neuen Ansatz entwickelt, bei dem die Bewertung mittels modellbasierter Techniken systematischer und objektiver durchgeführt werden kann [Schu12]. Dabei werden, wie in Abbildung 1 dargestellt, sowohl die Kontrollpunkte, d.h. die Anforderungen des TPI NEXT®-Modells an einen Testprozess, als auch der zu bewertende Testprozess selbst in der formalen Modellierungssprache BPMN [Om11] abgebildet. Während die Abbildung des Testprozesses zu einem kompletten Prozessmodell führt, werden Kontrollpunkte durch Modellfragmente dargestellt, die Geschäftsprozess-Muster genannt werden. Hierdurch wird die bisherige subjektive, erfahrungsbasierte Bewertung durch einen systematischen und visuellen Vergleich ersetzt, bei dem die Geschäftsprozess-Muster im Testprozesmodell gesucht werden. Zwar bleibt dieser systematische Vergleich weiterhin ein manueller Prozess, der allerdings durch den Einsatz von formalen Beschreibungssprachen und einem modellbasierten Verfahren die Schwächen eines rein subjektiven, erfahrungsbasierten Verfahrens überwindet.

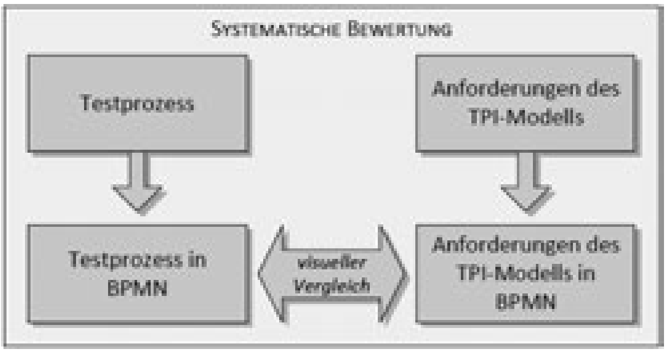


Abbildung 1: Überblick des Ansatzes

In den nächsten Abschnitten erläutern wir, wie die Kontrollpunkte des TPI NEXT®-Modells systematisch unter Verwendung der Modellierungssprache BPMN in Geschäftsprozess-Muster überführt werden (Abschnitt 3) und wie die modellbasierte Prozessbewertung mittels Geschäftsprozess-Mustern funktioniert (Abschnitt 4). Abschnitt 5 berichtet über die Evaluierung der Anwendbarkeit unseres Ansatzes mittels eines realen Fallbeispiels. Im Fazit schildern wir einen Ausblick, welche weiteren Möglichkeiten der modellbasierte Ansatz verspricht.

2. Verwandte Arbeiten

Es existieren bereits wissenschaftliche Arbeiten, die sich mit der Formalisierung und Qualitätssicherung von Geschäftsprozessen beschäftigt haben. Wohead et al. haben in [Wo06] Geschäftsprozess-Muster entwickelt, aus denen Workflows in BPMN erfolgreich aufgebaut werden können. Wie in unserer Arbeit steht auch bei Wohead et al. die Nutzung von Geschäftsprozess-Mustern im Vordergrund, allerdings als Mittel der konstruktiven Qualitätssicherung bei der Entwicklung von Prozessmodellen.

Die Dissertationen von Alexander Förster [Fö09] und Jens Müller [Mü10] entwickeln ein Konzept zur formalen Verifikation von Anforderungen an einen Geschäftsprozesses. Dazu modellieren sie den Geschäftsprozess und erstellen eine oder mehrere domänenspezifische Sprachen, um die Anforderungen an den Geschäftsprozess ebenfalls zu modellieren. Die daraus resultierenden formalen Modelle des Geschäftsprozesses und der Anforderungen werden dann in werkzeugspezifische Sprachen übersetzt und durch ein Model Checking-Werkzeug verifiziert. Als grundlegende Modellierungssprache verwendet Alexander Förster Aktivitätsdiagramme, Jens Müller die Modellierungssprache BPMN.

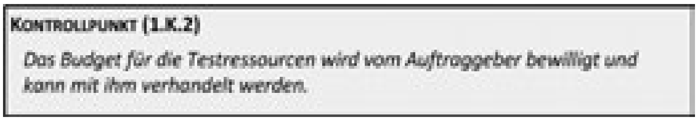
Unsere Arbeit ist durch diese beiden Dissertationen inspiriert, verfolgt aber im Gegensatz zu deren durchgängig formalen Ansätzen einen eher pragmatischen, in der Praxis einsetzbaren Ansatz. Hierdurch werden typische Probleme des formalen Ansatzes wie existierende Unterschiede der verwendeten Begriffe (Ontologien) oder das Abstraktions- und Verfeinerungsniveau von Prozessmodell und Prozessmustern überwunden und durch manuelle Tätigkeiten des Prüfers gelöst.

In unserem Ansatz verwenden wir für die Modellierung von Prozessen und Kontrollpunkten ebenfalls die Modellierungssprache BPMN. Unser Ansatz bezieht sich explizit auf Testprozesse und die durch das TPI NEXT®-Modell gegebenen Anforderungen an sie. Die Arbeit ist somit ein erster Schritt zu einer systematischen, letztendlich automatischen Bewertung nach TPI NEXT®.

3. Modellierung mittels Geschäftsprozess-Mustern

Um einen Testprozess modellbasiert mit den Anforderungen des TPI NEXT®-Modells vergleichen zu können, müssen sowohl der Testprozess selbst, als auch die durch das TPI NEXT®-Modell vorgegebenen Anforderungen an den Testprozess in BPMN modelliert werden. Im Folgenden wird die Modellierung der Kontrollpunkte als sogenannte Geschäftsprozess-Muster beschrieben.

Die für die Klassifizierung des Testprozesses ausschlaggebenden Anforderungen im TPI NEXT®-Modell sind durch die Kontrollpunkte beschrieben. Ein Kontrollpunkt ist eine informelle Aussage, die bei einer Bewertung mit *erfüllt* oder *nicht erfüllt* klassifiziert wird (Abbildung 2). Jeder Kontrollpunkt hat eine eindeutige ID, die die Nummerierung des Kernbereichs, die erste Buchstabe des Reifegrads und die Nummerierung des Kontrollpunkts enthält. Abbildung 2 zeigt den zweiten Kontrollpunkt im Kernbereich „1. Engagement der Stakeholder" im Reifegrad „Kontrolliert (K)" (vgl. auch Abbildung 9). Aus der Klassifizierung der einzelnen Kontrollpunkte aus einem Kernbereich resultiert die Reife des zu bewertenden Testprozesses. Folglich müssen die Kontrollpunkte in BPMN modelliert werden, um eine systematische Bewertung - anhand des Vergleichs zweier Modelle - durchführen zu können.



KONTROLLPUNKT (1.K.2)

Das Budget für die Testressourcen wird vom Auftraggeber bewilligt und kann mit ihm verhandelt werden.

Abbildung 2: Ein Kontrollpunkt des TPI NEXT®-Modells, welcher im Zuge einer Bewertung als *erfüllt* oder *nicht erfüllt* klassifiziert wird.

Um eine einheitliche und möglichst äquivalente Darstellung der informellen Kontrollpunkte in der formalen Sprache BPMN zu erhalten, wurde ein systematischer Ansatz zur Transformation entwickelt. Dazu wurden die Kontrollpunkte zunächst syntaktisch und semantisch analysiert. In dieser Analyse sind die in Abbildung 3 dargestellten Merkmale identifiziert worden, die den Verlauf und die Organisation eines Testprozesses charakterisieren, wie z.B. Teilnehmer, Informationsfluss und Zustand.

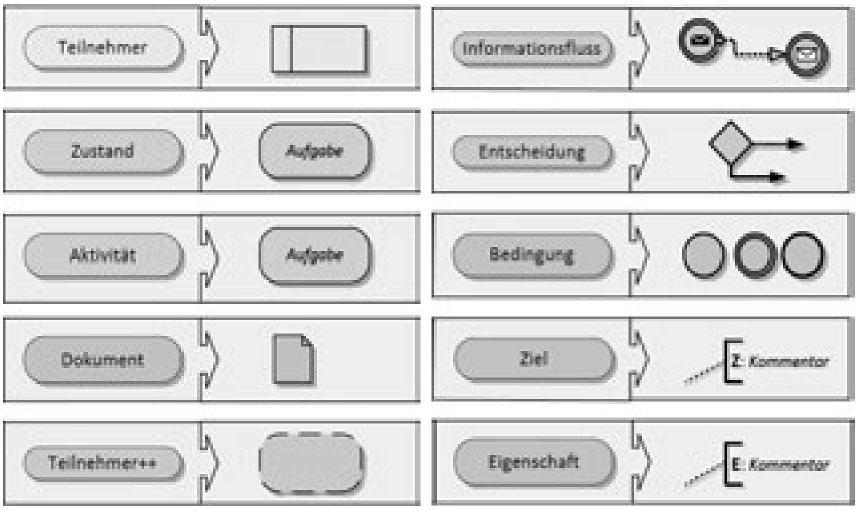


Abbildung 3: Transformationsrichtlinien, die sprachliche Merkmale der Kontrollpunkte mit BPMN-Sprachelementen verknüpfen.

Für die in der Analyse identifizierten Merkmale sind daraufhin repräsentative Elemente der Modellierungssprache BPMN hergeleitet worden, wodurch Transformationsrichtlinien entstanden sind. Abbildung 3 visualisiert die Herleitung der passenden BPMN-Sprachelemente für die Merkmale. Beispielsweise wird für das Merkmal *Teilnehmer* das BPMN-Sprachelement *Lane* hergeleitet. Eine vollständige Beschreibung der Herleitung der BPMN-Sprachelemente für die TPI-Merkmale befindet sich in [Schu12].

Mit Hilfe solcher Transformationsrichtlinien kann eine systematische Transformation der Kontrollpunkte nach BPMN durchgeführt werden. Dazu sind für jeden Kontrollpunkt die folgenden drei Schritte anzuwenden (Abbildung 4):

1. Identifizieren der in dem Kontrollpunkt vorhandenen Merkmale.
2. Herleiten der BPMN-Sprachelemente für die identifizierten Merkmale anhand der Transformationsrichtlinien.
3. Verknüpfung der BPMN-Sprachelemente entsprechend der Informationen des Kontrollpunkts.

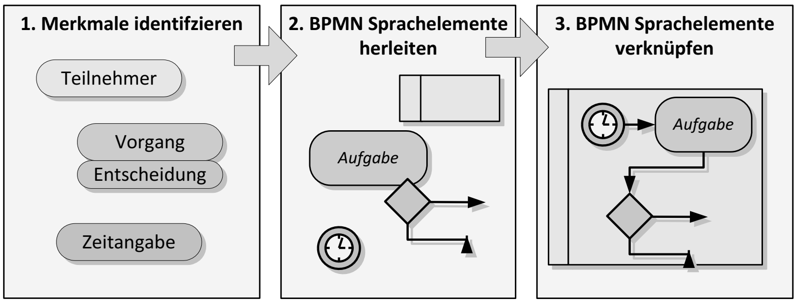

1. Merkmale identifzieren
Teilnehmer
Vorgang
Entscheidung
Zeitangabe
2. BPMN Sprachelemente herleiten
Aufgabe
3. BPMN Sprachelemente verknüpfen
Aufgabe

Abbildung 4: Die drei Schritte zur Transformation der Kontrollpunkte in Geschäftsprozess-Muster.

Da die Kontrollpunkte keinen zusammenhängenden Testprozess beschreiben, sondern jeder Kontrollpunkt punktuelle Anforderungen an diesen darstellt, entstehen bei der Transformation in BPMN lediglich Fragmente eines Testprozesses. Jedes Fragment stellt dabei die Anforderungen von genau einem Kontrollpunkt dar. Bei einer systematischen Bewertung wird ein derartiges Fragment wie ein Musterbeispiel mit dem Testprozess verglichen, weswegen wir die Fragmente als *Geschäftsprozess-Muster* nennen.

In Abbildung 5 wird beispielhaft gezeigt, wie die systematische Transformation eines Kontrollpunktes (01.K.2) in ein Geschäftsprozess-Muster durchgeführt wird. Zur Veranschaulichung erläutern wir einen Teil der zu identifizierenden Merkmale sowie deren Transformation. Im ersten Schritt werden die Merkmale des Kontrollpunktes identifiziert. Der Kontrollpunkt nennt den *Auftraggeber* als Teilnehmer im Testprozess. Das Verb *bewilligen* ist in der Verbform Vorgangspassiv und somit ein Indikator für das Merkmal *Aktivität*. Des Weiteren impliziert dieses Verb das Merkmal *Entscheidung*.

Im nächsten Schritt können anhand der Transformationsrichtlinien die für das Geschäftsprozess-Muster benötigten Sprachelemente in BPMN bestimmt werden. Die Richtlinie des Merkmals *Teilnehmer* führt zu einer Lane, welche mit *Auftraggeber* beschriftet wird. Aus dem Merkmal *Aktivität* wird das Aufgabe-Sprachelement *Budget für Testressourcen bewilligen* hergeleitet. Aus dem Merkmal *Entscheidung* resultiert das exklusive Gateway.

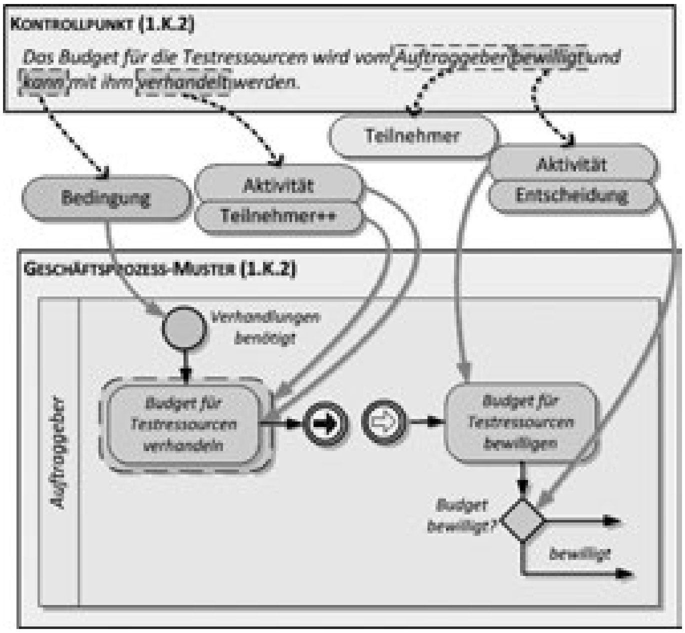

Kontrollpunkt (1.K.2)
Das Budget für die Testressourcen wird vom Auftraggeber bewilligt und kann mit ihm verhandelt werden.
Teilnehmer
Bedingung
Aktivität
Teilnehmer++
Aktivität
Entscheidung
Geschäftsprozess-Muster (1.K.2)
Auftraggeber
Verhandlungen benötigt
Budget für Testressourcen verhandeln
Budget für Testressourcen bewilligen
Budget bewilligt?
bewilligt

Abbildung 5: Transformation eines Kontrollpunktes in ein Geschäftsprozess-Muster.

Im letzten Schritt werden die identifizierten BPMN-Sprachelemente miteinander verknüpft und angeordnet. Die Aktivität *Budget für Testressourcen bewillige*n wird in der Lane des Auftraggebers platziert. Die Aktivität wird durch einen Sequenzfluss mit dem exklusiven Gateway *Budget bewilligt?* verbunden. Wird dies für alle Merkmale durchgeführt, resultiert das in Abbildung 5 dargestellte Geschäftsprozess-Muster. Eine vollständige Beschreibung ist in der Masterarbeit von Claudia Schumacher [Schu12] zu finden.

Die Kontrollpunkte des TPI NEXT$^{®}$-Modells können wie oben beschrieben systematisch in Geschäftsprozess-Muster transformiert werden. Auf diese Weise wurden im Rahmen der Arbeit 43 der 57 Kontrollpunkte des kontrollierten Reifegrads in Geschäftsprozess-Muster transformiert, was zur Veranschaulichung der beschriebenen Verfahren als ausreichend erachtet wurde (Anhang B in [Schu12] enthält eine vollständige Liste der erstellten Geschäftsprozess-Muster). Korrektheit und Vollständigkeit der einzelnen Muster werden durch das beschriebene Verfahren mit Hilfe der Transformationsrichtlinien sicher gestellt, zudem wurden die bisher erstellten Muster von TPI-Experten durch Review überprüft. Bis auf wenige Ausnahmen, bei denen Kontrollpunkte keine der oben beschriebenen sprachlichen Merkmale aufweisen, lassen sich jedoch auch die übrigen Kontrollpunkte, auch die des effizienten und optimierenden Reifegrads, interpretieren und an Hand der Transformationsrichtlinien in Geschäftsprozess-Muster überführen.

Wie in der Abbildung 1 dargestellt, muss für einen visuellen Vergleich neben den Kontrollpunkten auch der zu bewertende Testprozess in BPMN modelliert sein. Diese Vorbedingung führt zu einer teilweisen Erweiterung des Ablaufs zur Vorbereitung einer Bewertung nach TPI NEXT®. Denn ist der zu bewertende Testprozess nicht in BPMN vorhanden, muss dieser zunächst modelliert werden. In diesem Papier werden wir aber diesen Teil des Prozesses nicht näher erläutern.

4. Prozessbewertung mittels Mustererkennung

Die modellbasierte Bewertung von Testprozessen unterscheidet sich stark von den klassischen TPI NEXT®-Assessments. Im klassischen Bewertungsprozess führen die TPI NEXT®-Prüfer Interviews und sichten Dokumente, um die Bewertung der Kontrollpunkte und damit die Bestimmung der Testprozessreife vornehmen zu können. Dieses Vorgehen ist in großem Maße informell und erfahrungsbasiert, erfordert einerseits ein enormes Verständnis für verschiedenartige Testprozesse sowie andererseits perfekte Kenntnis des TPI NEXT®-Modells und seiner Kontrollpunkte, die im Hinblick auf den analysierten Testprozess hin korrekt interpretiert werden müssen.

Im Gegensatz dazu ermöglicht die Modellierung der Kontrollpunkte in BPMN-konforme Geschäftsprozess-Muster einen ebenfalls in BPMN modellierten Testprozess systematisch durch einen visuellen Vergleich zu bewerten. Es ist dazu notwendig, die BPMN-Darstellung(en) des Testprozesses auf das Vorhandensein der entsprechenden Muster zu durchsuchen, und beim Auffinden eines Musters den Kontrollpunkt, der durch das Muster repräsentiert wird, als erfüllt zu markieren.

Die Realität liegt natürlich zwischen den beiden geschilderten Ausprägungen einer erfahrungsbasiert durchgeführten und einer systematischen Bewertung. Das bedeutet, dass zu einem gewissen Anteil in TPI NEXT®-Assessments immer manuelle, informelle Aktivitäten durch die TPI NEXT®-Prüfer durchzuführen sind, um eine systematische, modellbasierte Bewertung erst zu ermöglichen. Wie hoch diese Aufwände sind und wie stark demgegenüber die Einsparungen durch einen modellbasierten Vergleich sein können, hängt von verschiedenen Voraussetzungen ab.

- Soll der tatsächlich gelebte Testprozess bewertet werden oder nur der dokumentierte Testprozess (z.B. zur Prüfung der Prozessdokumentation auf Vollständigkeit oder wenn ein Testprozess von Grund auf neu aufgesetzt wird)?
- Wurde die vorhandene Prozessdokumentation in BPMN verfasst, oder ist eine Modellierung bzw. Übersetzung der Dokumentation in BPMN erforderlich?
- Für die Bewertung des gelebten Testprozesses müssen die in Interviews und sonstiger Dokumentation erfassten Informationen über den Testprozess ebenfalls in BPMN dargestellt werden, um systematisch mit den Geschäftsprozess-Mustern verglichen werden zu können.

- Es kann Kontrollpunkte des TPI NEXT®-Modells oder Elemente des zu bewertenden Testprozesses geben, die weiterhin erfahrungsbasiert bewertet werden müssen, um den Testprozess vollständig abzudecken.

Das generelle Vorgehen bei der modellbasierten Bewertung wird nach Klärung dieser Fragen durch den in der Abbildung 6 skizzierten Ablauf beschrieben:

1. Durch Mustervergleich wird überprüft, ob alle beschrifteten Sprachelemente eines Musters zusammenhängend im Testprozess enthalten sind. Ist dies nicht der Fall, ist auch der Kontrollpunkt nicht erfüllt. Bestehen Unklarheiten, werden zur Bewertung weitere Informationen benötigt.
2. Werden die beschrifteten Sprachelemente gefunden, dann werden ihre Verknüpfungen aber auch mit den unbeschrifteten Sprachelementen überprüft. Stimmen die Verknüpfungen nicht mit dem Muster überein, kann ein Kontrollpunkt nur zum Teil erfüllt werden.
3. Stimmen schließlich auch die Verknüpfungen der Sprachelemente im Testprozess mit denen des Musters überein, wird der Kontrollpunkt als erfüllt markiert, ansonsten als Teilweise erfüllt.

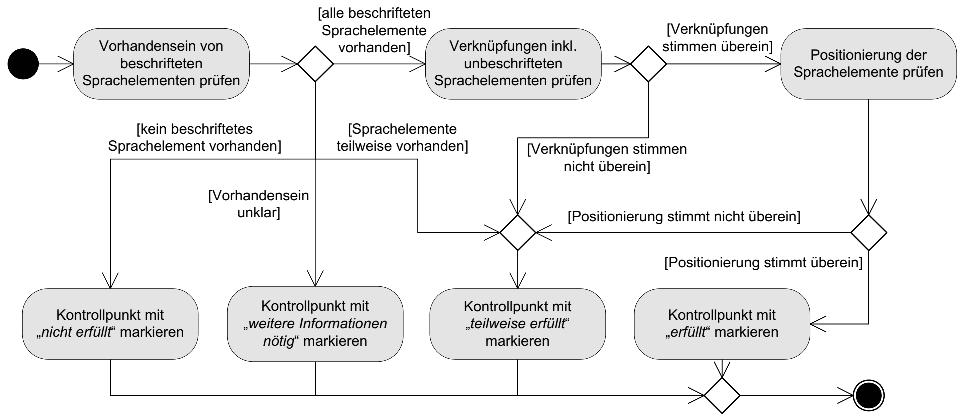

Vorhandensein von beschrifteten Sprachelementen prüfen
[alle beschrifteten Sprachelemente vorhanden]
Verknüpfungen inkl. unbeschrifteten Sprachelementen prüfen
[Verknüpfungen stimmen überein]
Positionierung der Sprachelemente prüfen
[kein beschriftetes Sprachelement vorhanden]
[Sprachelemente teilweise vorhanden]
[Verknüpfungen stimmen nicht überein]
[Vorhandensein unklar]
[Positionierung stimmt nicht überein]
[Positionierung stimmt überein]
Kontrollpunkt mit „nicht erfüllt“ markieren
Kontrollpunkt mit „weitere Informationen nötig“ markieren
Kontrollpunkt mit „teilweise erfüllt“ markieren
Kontrollpunkt mit „erfüllt“ markieren

Abbildung 6: Ablauf einer systematischen Bewertung

Das Ergebnis dieser Bewertung wird wie in klassischen TPI NEXT®-Assessments auch in Form einer Reifegrad-Matrix dargestellt (vgl. Abbildung 9). In der Praxis bleiben jedoch auch nach einer weitgehend systematischen Bewertung mittels Mustervergleich einige manuelle Aktivitäten zur Klärung erforderlich, für deren Systematisierung und Formalisierung weitere Methoden zu entwickeln wären.

- Bei Abweichungen in der Beschriftung von Testprozess-Modellen und Geschäftsprozess-Mustern können Fragen auftauchen weil z.B. unterschiedliche Beschriftungen dieselbe Bedeutung haben oder (was weitaus gravierender ist) gleich beschriftete Sprachelemente in Testprozessmodell und Mustern unterschiedliche Bedeutung haben.
- Der unterschiedliche Detaillierungsgrad von Testprozess und Mustern ist ebenfalls zu berücksichtigen. Zum Beispiel kann ein Sprachelement eines Musters durch mehrere Sprachelemente im Testprozess abgebildet sein oder mehrere Sprachelemente eines Musters werden im Testprozess zusammengefasst.

- Kontrollpunkte und damit auch die entsprechenden Geschäftsprozess-Muster können Anforderungen formulieren, die von verschiedenen Teilabläufen eines Testprozesses eingehalten werden müssen, d.h. ein Muster würde in diesem Fall mehrfach im Testprozess vorkommen.

Ein Beispiel für den Vergleich der Geschäftsprozess-Muster mit einem Testprozess zeigt Abbildung 7. Das Muster für den Kontrollpunkt 03.K.4 ("Bei Fehlernachtests und Regressionstests findet eine einfache Strategiefestlegung statt") ist teilweise enthalten. Das Festlegen einer Strategie für Regressionstests wird im Testprozess dargestellt, die geforderte Strategie für Fehlernachtests wird jedoch nicht dokumentiert.

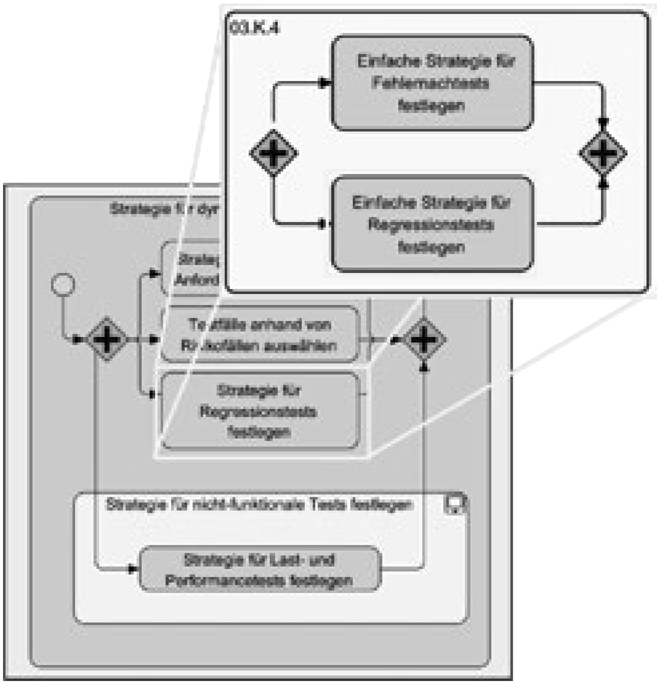

03.K.4
Einfache Strategie für Fehlernachtests festlegen
Einfache Strategie für Regressionstests festlegen
Strategie für dyn
Strateg Anford
Testfälle anhand von Risikofällen auswählen
Strategie für Regressionstests festlegen
Strategie für nicht-funktionale Tests festlegen
Strategie für Last- und Performancetests festlegen

Abbildung 7: Vergleich des Musters von Kontrollpunkt 03.K.4 mit einem Ausschnitt des Testprozesses. Das Muster ist teilweise enthalten.

Wird nun der dokumentierte Prozess bewertet, zeigt das Beispiel die Notwendigkeit einer Erweiterung der Dokumentation auf. Geht es jedoch um die Bewertung des reell gelebten Testprozesses, so ist es weiterhin Aufgabe der TPI NEXT®-Prüfer herauszufinden, ob z.B. die Strategiefestlegung für Fehlernachtests auch erfolgt, obwohl sie nicht explizit dokumentiert wurde oder ob der zweite Aspekt des Kontrollpunkts ggf. an anderer Stelle im Testprozess auftaucht. Beide Fälle würden für eine mögliche zukünftige Automatisierung weitere Entwicklungen der systematischen Bewertung erfordern.

5. Praxisbezug und Evaluierung

Um die Anwendbarkeit des Ansatzes zu zeigen, wurde ein bereits mit TPI NEXT® klassisch bewerteter Testprozess mit dem neuen Ansatz bewertet. Das anonymisierte Fallbeispiel bestand aus mehreren Dokumenten, die einen Testprozess, angelehnt an den fundamentalen Testprozess des ISTQB [SL10], bzgl. der Phasen Testplanung, Vorbereitung, Durchführung und Auswertung beschrieben. Ergänzend lagen Testkonzepte und Testrichtlinien vor. Zur Darstellung der Testaktivitäten wurden im Fallbeispiel als Modellierungssprache Ereignisgesteuerte Prozessketten (EPK) eingesetzt. Die EPK-Modelle sowie Informationen aus weiteren Dokumenten wurden in BPMN-Modelle übersetzt.

Abbildung 8 zeigt im Hintergrund ein Fragment des in BPMN modellierten Testprozesses. Des Weiteren wird das Geschäftsprozess-Muster zu Kontrollpunkt 04.K.3 („*Testaufgaben und Verantwortlichkeiten sind definiert und dokumentiert und einer Person oder Organisationseinheit zugewiesen*“) hervorgehoben, dessen Erfüllung hier beispielhaft geprüft wird. Die Bewertung bestand aus den im Abschnitt 4 geschilderten drei Schritten. Dabei wurde der semantische Vergleich der Knoten-Beschriftungen manuell durchgeführt. In diesem Beispiel wurde der Kontrollpunkt erfüllt.

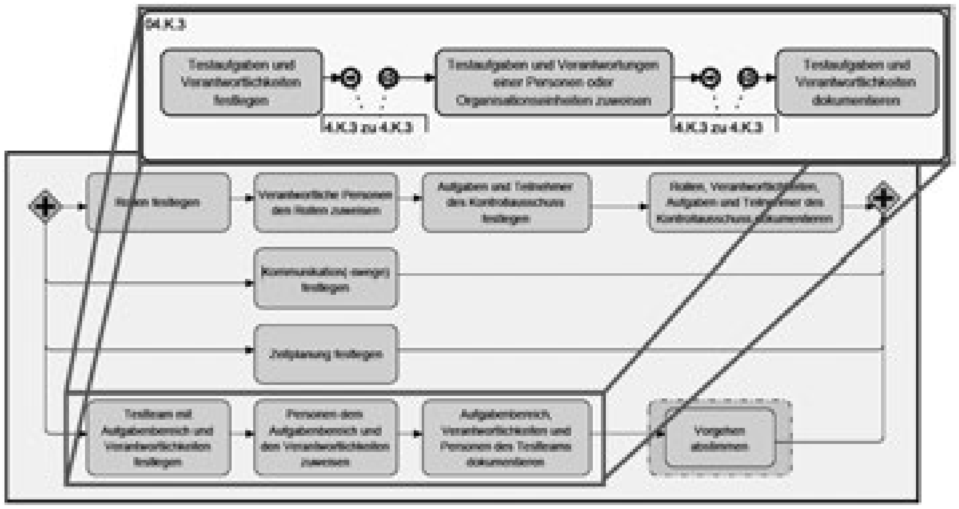

04.K.3
Testaufgaben und Verantwortlichkeiten festlegen
Testaufgaben und Verantwortungen einer Personen oder Organisationseinheiten zuweisen
Testaufgaben und Verantwortlichkeiten dokumentieren
4.K.3 zu 4.K.3
4.K.3 zu 4.K.3
Rollen festlegen
Verantwortliche Personen den Rollen zuweisen
Aufgaben und Teilnehmer des Kontrollausschuss festlegen
Rollen, Verantwortlichkeiten, Aufgaben und Teilnehmer des Kontrollausschuss dokumentieren
Kommunikation(-swege) festlegen
Zeitplanung festlegen
Testteam mit Aufgabenbereich und Verantwortlichkeiten festlegen
Personen dem Aufgabenbereich und den Verantwortlichkeiten zuweisen
Aufgabenbereich, Verantwortlichkeiten und Personen des Testteams dokumentieren
Vorgehen abstimmen

Abbildung 8: Die Übereinstimmung des Kontrollpunkts 04.K.3 und des Testprozessfragments

Das Ergebnis der Überprüfung der modellierten Kontrollpunkte des Reifegrads *Kontrolliert* ist in der Reifegrad-Matrix in Abbildung 9 dargestellt. Jede Zeile steht für einen der Kernbereiche, in die das TPI NEXT®-Modell den Testprozess gliedert. Jedes Feld in der Matrix entspricht einem Kontrollpunkt. Beispielweise entspricht das erste Feld unter dem Überschrift „Kontrolliert“ mit dem Zeichen „%“ dem Kontrollpunkt 01.K.1. Erste Ziffer „01“ entspricht dem Themenbereich, das in der jeweiligen Zeile angegeben ist. Die Buchstabe „K“ steht für Reifegrad „Kontrolliert“. Die letzte Ziffer „1“ steht für den ersten Kontrollpunkt für diesen Themenbereich. Als ein weiteres Beispiel entspricht das Feld rechts unten der Reifegrad-Matrix entspricht dem Kontrollpunkt 16.K.4.

Die Felder enthalten ein Zeichen, welches das Ergebnis der Überprüfung des Kontrollpunktes angibt. Ein Häkchen (✓) bedeutet, dass der Kontrollpunkt nach dem in Abbildung 6 beschriebenen Verfahren erfüllt wurde. Ein Prozent-Zeichen (%) bedeutet „teilweise erfüllt", da das Muster nur zum Teil im Testprozess enthalten ist. Das Frage-Zeichen (?) bedeutet, dass weitere Informationen für eine Entscheidung notwendig sind, zum Beispiel weil Beschriftungen nicht eindeutig sind (vgl. Abschnitt 4). Die mit dem Kreuz (×) gekennzeichnete Kontrollpunkte sind nicht erfüllt. Die in der Abbildungen 7 und 8 dargestellten Kontrollfragen 03.K.4 und 04.K.3 sind in der Reifegrad-Matrix entsprechend mit % und ✓ gekennzeichnet (s. hervorgehobene Felder in Abbildung 9).

			Kontrolliert			
1	Engagement der Stakeholder		%	×		×
2	Grad der Beteiligung		%	✓	?	?
3	Teststrategie		%	×	?	%
4	Testorganisation		%	✓	✓	×
5	Kommunikation		?	×	×	✓
6	Berichterstattung		×	?	✓	
7	Testprozessmanagement		✓	✓	×	%
8	Kostenschätzung und Planung		?	?	✓	%
9	Metriken		×	×	×	
10	Fehlermanagement		%	×	✓	
11	Testwaremanagement			×		
12	Methodisches Vorgehen					
13	Professionalität der Tester					
14	Testfalldesign		×	×	×	
15	Testwerkzeuge					
16	Testumgebung		×		×	×

Legende: ✓ erfüllt × nicht erfüllt % teilweise erfüllt ? weitere Informationen nötig

Abbildung 9: Kodierung der Reifegrad-Matrix nach Bewertung der Kontrollpunkte

Die abgebildete Reifegrad-Matrix visualisiert nicht das endgültige Ergebnis der Bewertung, sondern nur das Ergebnis nach dem visuellen Vergleich. Für das endgültige Ergebnis müssen nach TPI NEXT® alle Kontrollpunkte als *erfüllt* oder *nicht erfüllt* klassifiziert sein. Die Kontrollpunkte mit der Klassifikation *teilweise erfüllt* müssen daher von den Experten endgültig eingestuft werden. Diese ermitteln dazu weitere Informationen, die eine abschließende Klassifikation des Kontrollpunkts in *erfüllt* oder *nicht erfüllt* ermöglichen. Die Felder ohne Zeichen in Abbildung 9 repräsentieren die Kontrollpunkte, für die kein Geschäftsprozess-Muster erstellt wurde. Diese müssen ebenso nach dem klassischen Verfahren bewertet werden. Erst wenn dies abgeschlossen ist, steht das Endergebnis der systematischen Bewertung fest. Diese Weiterführung von dem Ergebnis des Vergleichs zum Ergebnis der Bewertung war im Zuge der Arbeit nicht relevant und ist aus diesem Grund nicht durchgeführt worden. Fest steht jedoch, dass durch die systematische Bewertung und die Klassifikation der Kontrollpunkte durch visuellen Vergleich der Bewertungsprozess auch für Nicht-Experten transparenter und nachvollziehbarer wird.

Als nächstes möchten wir erläutern, in wie weit die ursprünglichen Ziele der Arbeit bzgl. der Objektivität, Einfachheit und Effizienz erreicht wurden. Bzgl. der Objektivität der Prozessbewertung wurde erreicht, dass die Kontrollpunkte des TPI NEXT®-Modells zum größten Teil formal beschrieben werden können, was zu einer Reduktion der subjektiven Interpretationsmöglichkeiten führt. Zwar werden bei der Erstellung der Geschäftsprozess-Muster wegen der Mehrdeutigkeit der deutschen Sprache weiterhin subjektive Interpretationen benötigt, allerdings ist dies im Idealfall eine einmalige Aufgabe, die von Experten durchgeführt werden kann.

Der Bewertungsprozess wurde insbesondere für nicht-erfahrene TPI Next-Prüfer einfacher, da die Kriterien zur Erfüllung der Kontrollpunkte formal und objektiv festgelegt sind. Die Kontrollpunkte werden nun algorithmisch mit Hilfe des Mustervergleichs bewertet, womit der Prüfer einiges an Verantwortung bei der Bewertung abgibt.

Zur Evaluierung der Effizienz wurden noch keine empirischen Experimente durchgeführt. Wir stellen daher Effizienzbetrachtungen möglicher Szenarien theoretisch gegenüber. Bei der Bestimmung der Szenarien wurden die Verfügbarkeit von Prozessdokumentation und der Gegenstand der Bewertung berücksichtigt. Die Verfügbarkeit der Prozessdokumentation ruft drei Situationen hervor: (1) Prozessdokumentation in BPMN vorhanden, (2) Prozessdokumentation vorhanden, aber Prozess nicht in BPMN modelliert, (3) Prozessdokumentation nicht vorhanden

Als möglicher Bewertungsgegenstand kommen in Frage:

(1) Dokumentierter Testprozess (der Kunde möchte seine Prozessdokumentation mittels TPI NEXT® prüfen oder einen Testprozess neu aufsetzen, vgl. [Ew11])
(2) Gelebter Testprozess (der Kunde möchte den tatsächlich gelebten Testprozess bewerten oder prüfen, ob der dokumentierte Prozess eingehalten wird)

Aus der Kombination der zwei Aspekte konnten wir sechs Szenarien aufstellen (s. Abbildung 10), die sich bzgl. der Effizienzsteigerung unterscheiden. Wir haben diese Szenarien nach Effizienzunterschieden in den Bewertungsaktivitäten theoretisch untersucht und folgende Erkenntnisse erzielt:

Die modellbasierte Bewertung steigert die Effizienz von Assessments im Vergleich zum erfahrungsbasierten Ansatz,

- wenn der Testprozess bereits in BPMN modelliert ist, und man den definierten Prozess bewertet,
- wenn mehrere Bewertungs- und Verbesserungszyklen durchgelaufen werden,
- wenn ein Testprozess neu aufgesetzt wird, indem die Geschäftsprozess-Muster als Bausteine der Modellierung verwendet werden.

Die modellbasierte Bewertung reduziert die Effizienz von Assessments im Vergleich zum erfahrungsbasierten Ansatz,

- wenn der Testprozess nicht in BPMN oder gar nicht modelliert ist, und der Kunde das auch nicht will
- wenn nur eine einzige Bewertung des gelebten Testprozesses durchgeführt wird.

	Dokumentation in BPMN	Dokumentation nicht in BPMN	Dokumentation nicht vorhanden
Dokumentierten Testprozess bewerten	Effizienzgewinn wahrscheinlich.	Effizienzverlust durch Übersetzung der Prozessdokumentation in BPMN. Bei nachfolgenden Verbesserungszyklen Effizienzgewinn.	Effizienzgewinn bei von Grund auf neuem Testprozess. Effizienzverlust durch Erstellung der Prozessdokumentation
Gelebten Testprozess bewerten	Effizienzgewinn fraglich, wird mit mehreren Verbesserungszyklen wahrscheinlicher	Effizienzverlust durch Übersetzung der Prozessdokumentation in BPMN. Bei nachfolgenden Verbesserungszyklen Effizienzgewinn.	Effizienzverlust wahrscheinlich.

Abbildung 10: Theoretische Effizienzbetrachtungen bzgl. unterschiedlicher Szenarien

6. Fazit und Ausblick

Das Ziel der Zusammenarbeit zwischen dem s-lab und Sogeti war die Entwicklung einer einfachen, effizienten und vor allem objektiveren Methode zur Bewertung eines Testprozesses nach TPI NEXT®. Dafür haben wir einen Ansatz definiert, sowohl den zu bewertenden Testprozess als auch die Anforderungen an diesen in eine gemeinsame formale Darstellungsform zu überführen. Dies ermöglicht eine systematische Bewertung anhand eines visuellen Vergleichs. Als Darstellungsform wurde die Modellierungssprache BPMN ausgewählt, welche durch die OMG standardisiert ist und sich immer weiter durchsetzt.

Eine Evaluierung der Anwendbarkeit im Rahmen einer Fallstudie hat gezeigt, dass eine effiziente systematische Bewertung durch einen visuellen Vergleich der Geschäftsprozess-Muster mit dem Testprozess des Fallbeispiels möglich ist. Weiterhin wurde gezeigt, dass das Konzept eine objektivere Bewertung als zuvor ermöglicht, da die Bewertung anhand festgelegter Kriterien auf der Basis des Vergleichs zwei formaler Modelle erfolgt. Die Kriterien erleichtern ebenfalls die Entscheidung darüber, wie ein Kontrollpunkt klassifiziert wird. Ein Effizienzgewinn mit der modellbasierten Bewertung wird nur dann erwartet, wenn der Testprozess bereits in BPMN modelliert ist oder die Bewertung wiederholt durchgeführt wird. Allerdings gibt es noch weitere Vorteile, die aus der Modellierung des Testprozesses sowie der Klassifizierung entstehen. Wie die systematische Bewertung im Zuge einer praktischen TPI NEXT®-Bewertung eingesetzt werden kann, um diese neben der systematischen Bewertung zu vereinfachen, wird ebenfalls aufgezeigt.

Die Formalisierung der Anforderungen sowie die Beschreibung eines Vorgehens zur systematischen Bewertung können als ein erster Schritt für eine zukünftig automatisierte Bewertung von Testprozessen mit dem TPI NEXT®-Modell angesehen werden. Um eine

Automatisierung zu ermöglichen, müssen jedoch noch einige Herausforderungen gelöst werden. Eines der Hauptprobleme ist, die in verschiedenen Situationen benötigte Interpretation durch den Experten (wenn beispielsweise bei einem Vergleich eine Übereinstimmung zu vermuten ist) nicht eindeutig bestimmt werden kann. Hier könnten Listen von synonymen Beschriftungen verwendet oder das semantische Web (bzw. Ansätze zu dessen Informationsverarbeitung) genutzt werden. Dieser Lösungsansatz wurde bereits in der Dissertation von Jens Müller [Mü10] angeregt, aber nicht ausgeführt. Außerdem bieten die Ansätze von Alexander Förster [Fö09] und Lial Khaluf [KGE11] weitere Erweiterungsmöglichkeiten für unseren Ansatz, um die zeitlichen und logischen Beziehungen zwischen den Kontrollpunkten mittels temporaler Logik auszudrücken und diese in die automatisierte Bewertung einzubinden.

Die Arbeit zeigt einerseits, wie eine industriell im Bereich der Softwareprozessoptimierung entwickelte, vorwiegend praktisch begründete Methode wissenschaftlich fundiert aufbereitet und erweitert wird. Die Ergebnisse der wissenschaftlichen Arbeit fließen andererseits in die praktische Anwendung bei TPI NEXT®-Assessments zurück und bringen durch die Verknüpfung zu BPMN neue hilfreiche Aspekte für die praktische Arbeit als TPI NEXT®-Prüfer.

Literaturverzeichnis

[Ew11] van Ewijk, A. et al.: „TPI NEXT: Geschäftsbasierte Verbesserung des Testprozesses“, dpunkt.Verlag GmbH, 2011.

[Fö09] Förster, A.: “Pattern based business process design and verification”, Universität Paderborn, Dissertation, 2009.

[Hu95] Humphrey, W.: “Managing the Software Process”, Addison-Wesley, 1995.

[KGE11] Khaluf, L.; Gerth, C.; Engels, G.: “Pattern-Based Modeling and Formalizing of Business Process Quality Constraints”, In: H. Mouratidis, C. Rolland (eds.): “Proceedings of the 23rd International Conference on Advanced Information System Engineering (CAiSE'11)”, Springer (Berlin/Heidelberg), LNCS, vol. 6741, pp. 521-535, 2011.

[Mü10] Müller, J.: „Strukturbasierte Verifikation von BPMN-Modellen“, Eberhard-Karls-Universität Tübingen, Dissertation, 2010.

[Om11] Object Manamegement Group (OMG): “Specification: Business Process Model and Notation”, 2011.

[Schu12] Schumacher, C.: „Systematische Bewertung von Testprozessen nach TPI NEXT® mit Geschäftsprozess-Mustern“, Masterarbeit, Universität Paderborn, 2012.

[SL10] Spillner, A.; Linz, T.: Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester- Foundation Level nach ISTQB-Standard. Dpunkt.Verlag GmbH, 2010.

[Wo06] Wohead, P. et al.: “Pattern-based Anaysis of BPMN - an extensive evaluation of the Control-flow, the Data and the Resource Perspectives”, Queensland University of Technology, 2006.

Kontextabhängige Best Practices in einer Beschleunigten Softwaretechnik

Markus Voß, Martin Lehmann,
Daniel Contag, Georg Fahlbusch, Thorsten Kimmig

Accso – Accelerated Solutions GmbH
Berliner Allee 58, 64295 Darmstadt
{voss|lehmann|contag|fahlbusch|kimmig}@accso.de

und

Hochschule Darmstadt (h_da), Fachbereich für Informatik

Abstract: Im Rahmen einer Beschleunigten Softwaretechnik zur Steigerung der Effizienz in der Softwareentwicklung spielen aus Projekterfahrung abgeleitete Best Practices eine wesentliche Rolle, die Methoden des Software-Engineering konkretisieren und vielfach erst anwendbar machen. Zur Sicherstellung eines für das jeweilige Projekt angemessenen Vorgehens sind die vom konkreten Projektkontext abhängigen Best Practices besonders interessant. Aus diesem Grund führen wir derzeit ein Forschungsprojekt durch, in dem solche kontextabhängigen Best Practices erhoben und näher untersucht werden. In diesem Beitrag führen wir kurz in die Beschleunigte Softwaretechnik ein, definieren Projektkontext und Einflussfaktoren, strukturieren Best Practices inhaltlich, berichten in groben Zügen über das Forschungsprojekt und zeigen beispielhaft einige Best Practices.

1. Einleitung

Eine von der Accso GmbH in diesem Jahr beauftragte Umfrage unter IT-Entscheidern [Fr12] hat bestätigt, was wir alle schon länger ahnten: Allein der Deutschen Wirtschaft entsteht jährlich ein Milliarden-Schaden aufgrund von Ineffizienzen in der Softwareentwicklung. Die Effizienz in industriellen Softwareprojekten zu steigern, ist und bleibt also ein wichtiges Ziel.

Effizienz bedeutet in diesem Zusammenhang erst einmal klassisch das Verhältnis vom *Ertrag* eines Softwareprojekts – den Projektergebnissen – zum *Aufwand* der benötigt wird, diese Projektergebnisse zu erstellen, d.h. insbesondere den benötigten personellen Kapazitäten. Dabei ergibt sich der Ertrag eines Softwareprojekts aber nicht quantitativ z.B. über die Lines of Code oder die Menge der erstellten Dokumentation, sondern qualitativ über den Grad der Erfüllung der Anforderungen. So können im Detail unterschiedliche Projektergebnisse, die insbesondere einen unterschiedlich hohen Erstellungsaufwand benötigen, vom Ertrag her vergleichbar sein. In diesem Sinne kann die *Effektivität* des Projektvorgehens, nämlich von verschiedenen Ergebnisvarianten mit vergleichbarer Anforderungsabdeckung die mit dem niedrigsten Aufwand zu wählen, unter den hier zugrunde gelegten Begriff der Effizienz subsumiert werden.

Zwei Ansätze werden heutzutage üblicherweise mit einer Steigerung der Effizienz in der Softwareentwicklung assoziiert:

- **Software-Industrialisierung**: Hier wird die Effizienz dadurch gesteigert, dass Prozesse und Tätigkeiten der Softwareentwicklung stark standardisiert, arbeitsteilig (sprich mit einem hohen Grad an Spezialisierung) bearbeitet und soweit wie möglich automatisiert werden. Man orientiert sich also an einem Grundverständnis von effizientem Vorgehen, wie es aus der Produktion schon lange bekannt ist. Voraussetzung dafür sind jedoch eine weitgehende Planung des Entwicklungsprojektes mit exakt definierten Tätigkeiten und Verantwortlichkeiten (z.B. in Form von Rollen) sowie eine sehr weitgehende Spezifikation des zu entwickelnden Produktes. Diese Denkweise passt auch sehr gut zu einem klassischen „generalstabsmäßigen“ Verständnis vom Management solcher Projekte.
- **Agile Methoden**: Hier geht man davon aus, dass sich Projekte im Vorfeld nicht komplett planen und Produkte nicht komplett spezifizieren lassen. Vielmehr muss man sich auf Veränderungen so einfach und schnell wie möglich einstellen können, was durch ein leichtgewichtiges Lean Management unterstützt wird. Effizienz gewinnt man hier vor allem durch die Vermeidung von unangemessen komplexen Prozessen und Organisationsformen und durch die Optimierung der Nutzung der Potenziale des gesamten Teams durch weitgehende Selbstorganisation und ein hohes Maß an Vertrauen.

Allein diese plakative Gegenüberstellung mit ihren teilweise widersprüchlichen Grundannahmen zeigt schon, dass eine allgemeine Antwort auf die Forderung nach Effizienzsteigerung in der Softwareentwicklung nur schwer zu finden ist. Planung von Softwareprojekten ist gut, aber nur so weit, wie halbwegs verlässliche Planannahmen getroffen werden können und die Beschäftigung mit Planung und Controlling nicht zu Lasten der kreativen Lösungsfindung und Umsetzung geht. Spezifikation von Softwareprodukten ist gut, aber nur so weit, wie sich nicht währenddessen die Anforderungen unbeachtet ändern können oder Dinge spezifiziert werden, die nicht um ein vielfaches effizienter iterativ entwickelt werden könnten. Spezialisierung und Projektrollen sind gut, aber nur so weit, wie sie nicht zu Engstirnigkeit, Verweigerung der Übernahme von Verantwortung oder ausufernden Abstimmungsaufwänden führen. Und auch Standardisierung und Automatisierung sind gut, aber nur so lange, wie dies wirklich nur Routineprozesse betrifft und die so wichtige Kreativität nicht verloren geht.

Aus diesen Überlegungen heraus stellen wir den Begriff der **Angemessenheit** in das Zentrum unserer Überlegungen zur Effizienz in der Softwareentwicklung. Angemessenes Handeln heißt, aus der Menge möglicher Handlungsalternativen und Mittel diejenigen auszuwählen, die zu einem effizienzoptimierten Trade-Off führen. Die etablierte moderne Softwaretechnik als Menge aller uns heutzutage zur Softwareentwicklung zur Verfügung stehenden Verfahren, Methoden, Werkzeuge, Handlungsanweisungen ist von sich aus erst einmal weder effizient noch ineffizient, dient aber als Quelle und Werkzeugkasten Erst wenn die Softwaretechnik um konstruktive Mittel ergänzt wird, Entscheidungen bei der Auswahl systematisch treffen zu können, kann man davon sprechen, dass die Softwaretechnik selber effizient ist. Wir nennen das *Beschleunigte Softwaretechnik*.

2. Beschleunigte Softwaretechnik

Beschleunigte Softwaretechnik (kurz BeST) ist ein Konzept zur effizienten Softwareentwicklung, welches sich durch folgende Eigenschaften auszeichnet:

- **Ganzheitlichkeit**: Nicht nur das Vorgehensmodell, sondern *alle* Aspekte von Softwaretechnik wie Methodik (z.B. für Analyse bzw. Spezifikation), Software-Architektur, Entwicklungsthemen wie die Wahl der Programmiersprache oder der Einsatz von Komponenten und Frameworks, das Projektmanagement sowie alle Aspekte der Projektumgebung mit den einzusetzenden Technologien und Werkzeugen werden hinsichtlich ihrer Bedeutung für die Effizienz betrachtet. Nur wenn alle Aspekte ganzheitlich optimiert werden, kann man auf eine relevante Effizienzsteigerung hoffen. In [LL10] wird das als globale Optimierung bezeichnet.
- **Angemessenheit**: In allen genannten Aspekten der Softwaretechnik gibt es vielfältige Freiheitsgrade. Kunden- bzw. vorhabenspezifisch wird jeweils der optimale Trade-Off zwischen „so einfach und schlank wie möglich“ und „so elaboriert und strikt wie nötig“ ermittelt und das Gesamtvorgehen darauf zugeschnitten.
- **Skill-Orientierung**: Das Know-how im Entwicklungsteam hat nach unserer Erfahrung von allen Wirkfaktoren den größten Einfluss auf Kosten und Aufwände. Die Ausbildung der benötigten Fähigkeiten für diesen anspruchsvollen Ansatz ist integraler Bestandteil des Konzepts.

2.1 Ganzheitlichkeit: Die fachlichen Domänen

Zur Sicherstellung der Ganzheitlichkeit des Ansatzes werden alle softwaretechnischen Mittel im sogenannten *BeST-Framework* in fachliche Domänen eingeteilt – von der Analyse bis zum Test inklusive querschnittlicher Themen des Projektmanagements und Aspekten der Umgebung der Softwareentwicklung (vgl. Abb. 1 links unten).

Die Inhalte der einzelnen Domänen sind dabei vergleichbar zu etablierten Architektur-Frameworks wie [VW10] oder [EH08] zielorientiert aufgebaut. Aus den *Zielen* einer Domäne („*Warum* ist etwas zu tun?“) leiten sich *Aufgaben* ab, die nötig sind, diese Ziele zu erreichen („*Was* ist zu tun?“). Zur Umsetzung der unterschiedlichen Aufgaben gibt es *Methoden*, die Mittel und Wege aufzeigen, die Aufgabe zu lösen („*Wie* ist es zu tun?“). Dabei werden definierte Artefakte verwendet oder erstellt („*Womit* ist es zu tun?“). Ergänzt wird dieses Wissen durch einen strukturierten Überblick sowie Referenzen in Form von Literaturverweisen. Abb. 1 rechts veranschaulicht diese Strukturierung einer BeST-Domäne allgemein. Abb. 2 zeigt die Strukturierung beispielhaft für die Domäne Projektmanagement.

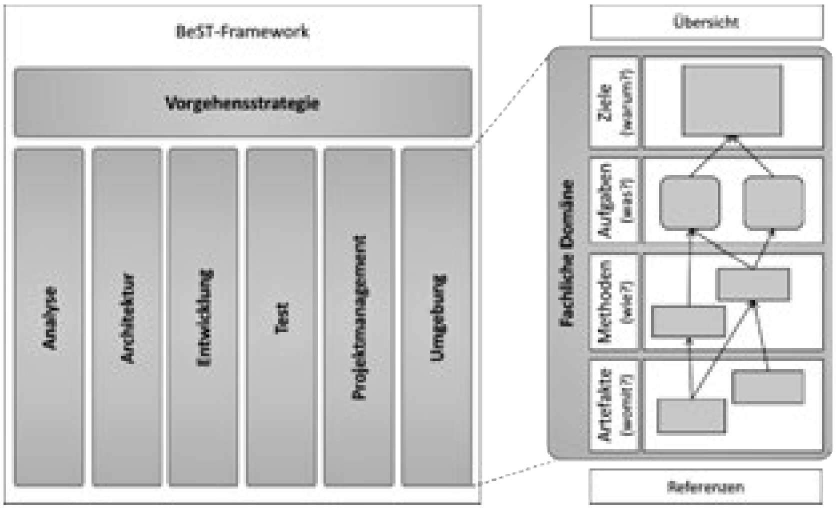


Abbildung 1: BeST-Framework – Überblick (links) und Strukturierung einer Domäne (rechts)

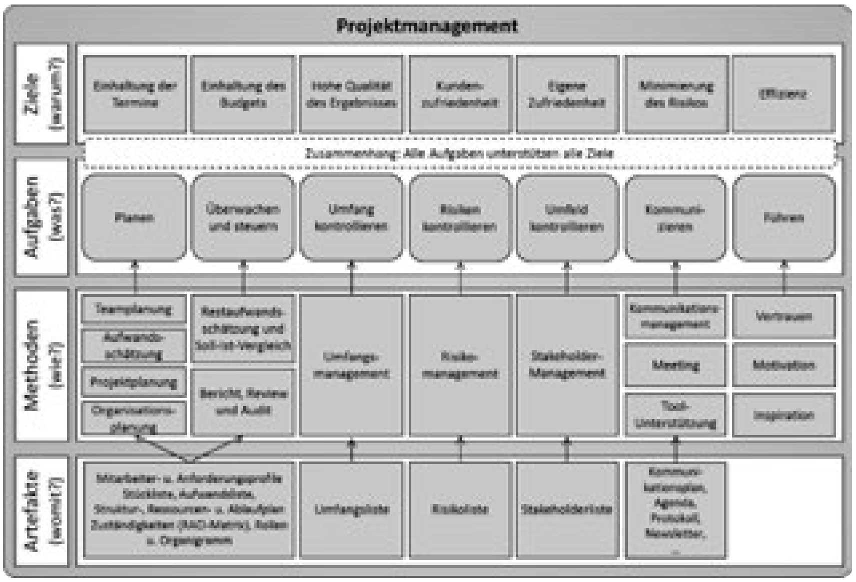


Abbildung 2: Strukturierung der fachlichen Domäne Projektmanagement (beispielhaft)

Entscheidend für die Steigerung der Vorgehenseffizienz ist nun aber nicht diese Strukturierung des Wissens allein. Vielmehr liefert diese den Rahmen für eine systematische Beschäftigung mit dem Thema Softwaretechnik. Um im konkreten Softwareprojekt tatsächlich eine Effizienzsteigerung zu erzielen, ist es darüber hinaus entscheidend, ganz konkrete **Best Practices** an die Hand zu bekommen. Diese Best Practices definieren, welche Methoden zur Bewältigung einer bestimmten Aufgabe *erfahrungsgemäß* ausgewählt werden sollen und wie genau diese Methoden erfahrungsgemäß anzuwenden sind.

Tatsächlich ist es so, dass vieles, was in der heutigen Softwaretechnik als Methode gehandelt wird und damit eigentlich eine handfeste Aussage zum „*Wie* ist es zu tun?“ machen sollte, nicht ohne weiteres im konkreten Umfeld auch (effizient) angewandt werden kann. Ein schönes Beispiel dafür ist das Project Management Body of Knowledge (PMBoK) des PMI [PMI08]. Methoden und Vorgehensweisen werden dort detailliert in Prozessen und Artefakten definiert (in der Strukturierung ähnlich zum Modell in Abb. 2). In diesem Bereich ist das PMBoK auch eine äußerst nützliche Referenz und selbstverständlich ist ein Projekt deutlich effizienter zu managen, wenn alle diese Standard-Prozesse und -Artefakte bekannt sind und nur implementiert und nicht neu erfunden werden müssen. Der Anspruch der Beschleunigten Softwaretechnik ist nun aber, zusätzlich zu den etablierten Methoden wie denen des PMBoK konkrete Handlungsanweisungen zum tatsächlichen Umgang mit den gegebenen Prozessen und Artefakten in Form von direkt anwendbaren Best Practices zu geben.

Ein plakatives Beispiel dafür ist die Erstellung des Projektstrukturplans und des Projektablaufplans. Das PMBoK benennt diese Artefakte und die Prozesse, diese zu erstellen. Ein unerfahrener Projektmanager läuft trotzdem Gefahr, insbesondere den Ablaufplan viel zu detailliert vorab zu planen und dementsprechend auch jede Veränderung im Ablauf im Plan nachziehen zu wollen. Und wenn er dann auch noch mit einem komplexen Werkzeug wie beispielsweise MS Project arbeitet, ist die Chance hoch, dass es zu spürbarer Ineffizienz kommt. Hilfreich ist in diesem Fall also eine einfache Best Practice der Art „Plane Deine Projektstruktur in Form der einzelnen Tasks (Backlog) möglichst genau, bleibe bei der Ablaufplanung aber auf einen eher groben Niveau, das insbesondere die Meilensteine berücksichtigt, ansonsten aber soweit wie möglich stabil bleibt“.

Kurz gesagt: Gegebene Methoden geben natürlich Hinweise, „Wie?“ etwas getan werden soll, oft liefern aber erst die Best Practices, die konkrete Praxiserfahrungen widerspiegeln, die Antwort auf die Frage nach dem „Wie genau?“. Beschleunigte Softwaretechnik umfasst dementsprechend eine umfangreiche Sammlung solcher Best Practices, die zu einer Effizienzsteigerung in Softwareprojekten beitragen können. Best Practices, die die Effizienz steigern und die Bestandteil der Beschleunigten Softwaretechnik (also von BeST) sind, bezeichnen wir im Folgenden als *BeST-Practices*.

Dabei gibt es zwei Arten von BeST-Practices:

- **Allgemeingültige** BeST-Practices gelten immer, unabhängig vom speziellen Projekt und dessen Gegebenheiten. Daher sind sie vielfach auch sehr bekannt und müssen „nur noch“ konsequent im Projekt berücksichtigt werden. Schöne Beispiele hierfür sind:
    - Vermeide die Aufstockung des Teams durch weitere Projektmitarbeiter, wenn das Projekt bereits verspätet ist, weil das sonst eher zu noch mehr Verspätung führt („Brooks‘ Law“ [Br75]).
    - Aufwandsschätzungen – egal nach welcher Methode – sollten immer von den fachlichen und technischen Experten und nicht von Managern erstellt werden.

- **Kontextabhängige** BeST-Practices gelten hingegen nur, wenn bestimmte Randbedingungen im Projekt gegeben sind. Gelten andere Randbedingungen, so können diese Practices auch kontraproduktiv sein.

Das führt uns zum Thema Angemessenheit und zur Domäne der Vorgehensstrategie.

2.2 Angemessenheit: Die Domäne der Vorgehensstrategie

Wesentlich zur Sicherung der Angemessenheit sind die projektindividuelle Auswahl und der projektindividuelle Zuschnitt gegebener Methoden und Techniken. BeST-Practices hierzu werden in der querschnittlichen Domäne der Vorgehensstrategie organisiert (vgl. Abb. 1 links oben). Die Domäne Vorgehensstrategie umfasst das Wissen darüber, wie – abhängig von den *Einflussfaktoren* des Projekts – das konkrete Vorgehensmodell für alle Aufgabenbereiche des Softwareprojekts bestimmt wird. Solche Einflussfaktoren können u.a. mit dem Projektauftrag, mit dem Kunden und seinen Anforderungen sowie mit den vertraglichen Randbedingungen zu tun haben. Weiterhin relevant sind z.B. das eingesetzte Projektteam, Vorgaben zur zu verwendenden Technologie und zur Architektur sowie jegliche Vorgaben zum Projektvorgehen selber. Weiter unten gehen wir näher darauf ein. Die Menge aller Einflussfaktoren nennen wir den *Projektkontext*.

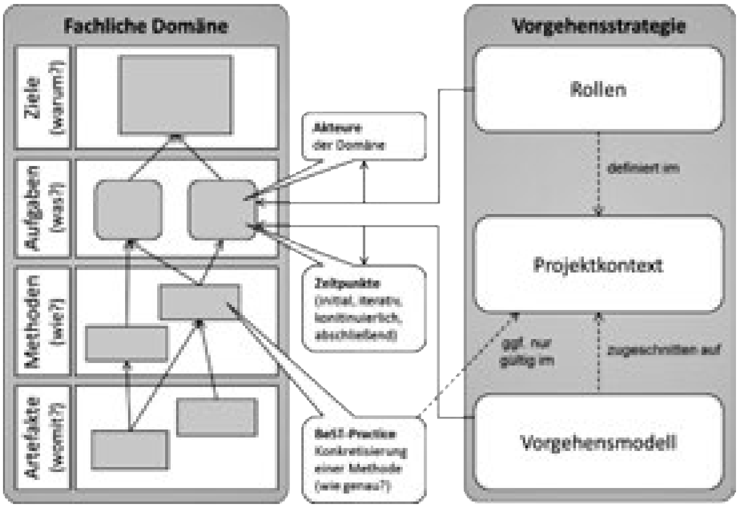


Abbildung 3: Zusammenhang zwischen den Domänen

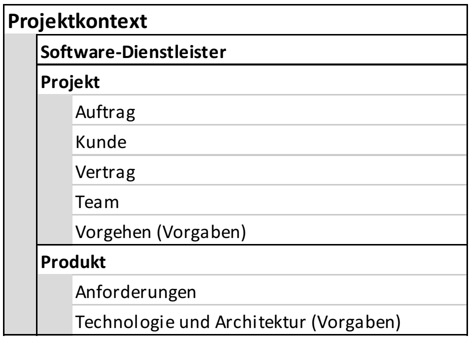
Die Domäne Vorgehensstrategie bietet den Rahmen dafür, dass *kontextabhängig* die passenden einzelnen Methoden und BeST-Practices aus den anderen Domänen ausgewählt und ggf. zugeschnitten sowie die dabei zu besetzenden Rollen festgelegt werden können. Die Aufbereitung dieses Wissens kann dabei ebenfalls in Form von BeST-Practices erfolgen - es handelt sich dabei um die oben bereits definierten kontextabhängigen BeST-Practices. Abb. 3 stellt diesen Zusammenhang zwischen den fachlichen Domänen und der Domäne der Vorgehensstrategie dar.

Im Folgenden widmen wir uns der Erarbeitung von Inhalten dieser Domäne Vorgehensstrategie. Dabei legen wir das aus Sicht der Accso GmbH sinnvolle Modell einer industriellen Software-Entwicklung in einem Auftraggeber/Auftragnehmer-Verhältnis aus Sicht des Software-Dienstleisters und Auftragnehmers zugrunde.

3. Projektkontext

Dass der Projektkontext einen wesentlichen Einfluss auf die mit einem Softwareprojekt verbundenen Kosten, Aufwände und Risiken und damit auch auf das Vorgehen hat, ist allgemein bekannt. So tauchen Einflussfaktoren beispielsweise in vielen der gängigen Methoden zur Aufwandsschätzung wie dem Constructive Cost Model (CoCoMo) [Bo81] oder der erweiterten Use-Case-Point Methode [Fr09] als explizite Faktoren bzw. Gewichte auf, mit denen die zu erwarteten Umsetzungsaufwände multipliziert werden, um den Spezifika des jeweiligen Projekts Rechnung zu tragen. Einige Arbeiten wie [CO12] erarbeiten elaborierte Referenz-Frameworks zur Strukturierung des Projektkontexts. Andere Arbeiten wie [SBV04] untersuchen die Auswirkung von Einflussfaktoren post-mortem nach Projektabschluss. Und es gibt auch allgemeine Untersuchungen zur Abhängigkeit zwischen dem Projektkontext und Auswahl bzw. Zuschnitt des Projektvorgehens wie beim Situational Method Engineering [WK92] oder beim Situational Software Engineering [EK11].

Unabhängig von der jeweils gewählten Strukturierung der in den einzelnen Ansätzen jeweils betrachteten Einflussfaktoren, lässt sich der Projektkontext dabei eigentlich immer in grundsätzliche Kategorien untergliedern. Auf oberster Ebene lassen sich die projektübergreifenden Einflussfaktoren in Form der Charakteristika des Software-Dienstleisters, der unterschiedliche Projekte für unterschiedliche Kunden durchführt, die projektbezogenen Einflussfaktoren und die produktbezogenen Einflussfaktoren unterscheiden. Projektbezogene, d.h. für das Projekt jeweils spezifische Einflussfaktoren lassen sich weiter unterteilen in auftragsbezogene, kundenbezogene, vertragsbezogene, teambezogene und vorgehensbezogene Einflussfaktoren. Produktbezogen spielen die Anforderungen selber eine Rolle (funktionale und nicht-funktionale) sowie Randbedingungen mit Vorgaben hinsichtlich Technologie und Architektur der zu entwickelnden Lösung. Abb. 4 zeigt diese Struktur.



Projektkontext		
	Software-Dienstleister	
	Projekt	
		Auftrag
		Kunde
		Vertrag
		Team
		Vorgehen (Vorgaben)
	Produkt	
		Anforderungen
		Technologie und Architektur (Vorgaben)

Abbildung 4: Top-Level Kategorisierung der Einflussfaktoren

Abb. 5 setzt diese Kategorien von Einflussfaktoren mit den anderen im Folgenden verwendeten Begriffen der Softwaretechnik in Beziehung.

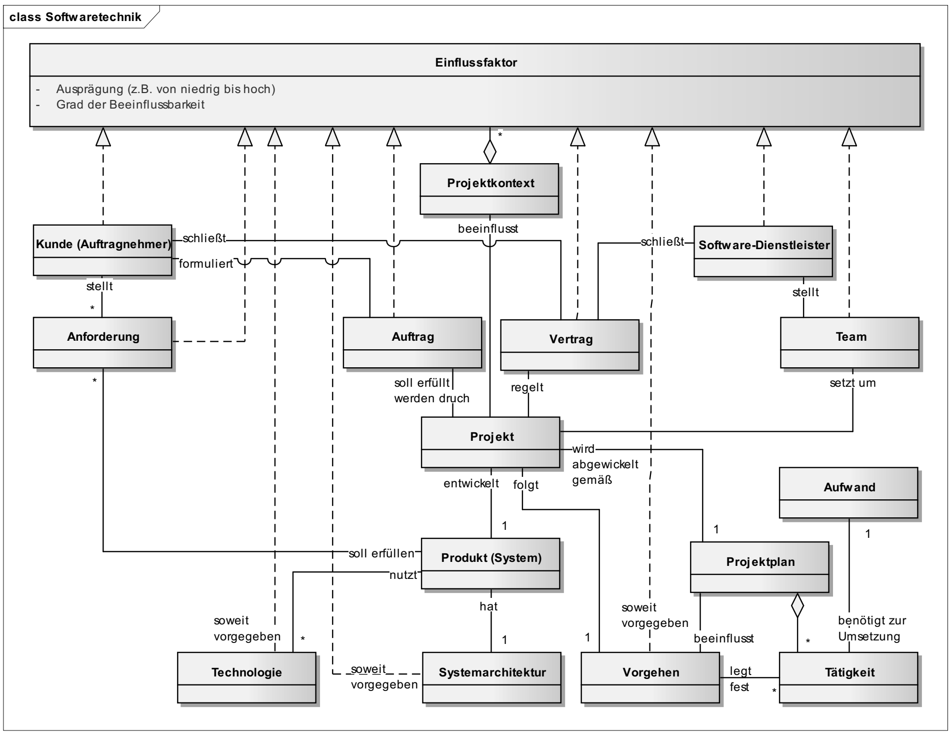


Abbildung 5: Verwendete Begriffe der Softwaretechnik

Der Kategorisierung lassen sich nun alle einzelnen konkreten Einflussfaktoren zuordnen. Dabei tritt jeder Einflussfaktor im konkreten Fall mit einer konkreten Ausprägung auf (z.B. niedrig – hoch) und zeichnet sich zudem durch seinen Grad der Beeinflussbarkeit durch die Handelnden im Projekt aus.

Bis zu diesem Punkt des Verständnisses der Rolle von Einflussfaktoren sind alle oben genannten Ansätze, das Vorgehen in Softwareprojekten projektkontextspezifisch festzulegen, im Grunde identisch. Spannend wird es nun, wenn man sich anschaut, was in etablierten Methoden und der Literatur an konkreten Ableitungen zum Vorgehen aus dieser Kenntnis der Einflussfaktoren zu finden ist. Hier endet die Betrachtung oft bei relativ allgemeinen Aussagen wie „In eher kleinen Projekten (=Einflussfaktor) sind agile Methoden besser geeignet als in großen“ oder „In Projekten mit der Vertragsform Festpreis (=Einflussfaktor) scheidet ein agiles Vorgehen nach Scrum aus“. Auch in spezifisch auf die benannte Fragestellung der kontextabhängigen Vorgehensweisen ausgerichteten Ansätzen wie [EK11] bleibt die Vorgehensempfehlung eher vage. Völlig zurecht werden hier zwar Zusammenhänge zwischen niedrigen bzw. hohen Ausprägungen von Einflussfaktoren wie Projektgröße, Anforderungsdynamik, Kundenkultur oder zu erwartende Kundenmitarbeit in Beziehung gesetzt zu niedrigen bzw. hohen Anforderungen an die Prozesse des Requirement Engineerings und des Qualitätsmanagements, die Nachhaltigkeit von Technologie und Architektur oder die Menge an Dokumentation. Die Ableitung konkret einsetzbarer Best Practices fehlt jedoch. Am konkretesten sind noch Ansätze wie [HE10], der zumindest im Bereich der Auswahl eines situationsgerechten Vorgehensmodells relativ konkret wird, oder Arbeiten wie [DL10], in denen konkret die Auswahl unter mehreren gegebenen agilen Managementmethoden thematisiert wird.

Keiner der uns bekannten Ansätze benennt jedoch explizite situationsgerechte Best Practices im obigen Sinne. Genau hier setzen wir mit der weiteren Erarbeitung einer Beschleunigten Softwaretechnik an und die Definition weiterer konkreter kontextabhängiger BeST-Practices, die noch nicht allgemein bekannt sind, ist Inhalt des derzeit laufenden Forschungsprojekts, welches weiter unten beschrieben wird. Zunächst wollen wir jedoch näher erläutern, was eine BeST-Practice inhaltlich ausmacht bzw. wie diese strukturiert beschrieben werden kann.

4. Strukturierte BeST-Practices

Wie bereits beschrieben basieren BeST-Practices auf bekannten Methoden bzw. den darin enthaltenen einzelnen durchzuführenden Prozessen und Tätigkeiten und konkretisieren diese. Damit sind zumeist auch die Artefakte beschrieben, die in den Tätigkeiten benötigt bzw. durch die Tätigkeiten erstellt werden. Eine BeST-Practice sorgt nun insoweit für eine Konkretisierung bzw. Präzisierung dieser Tätigkeiten, indem sie Zusatzinformationen über den reinen Umfang der Tätigkeit hinaus angibt. Eine BeST-Practice besteht aus den Teilen

- **Tätigkeit** im Sinne einer Referenz auf eine Tätigkeit aus einer bekannten Methode, z.B. „Spezifikation von Use-Cases“
- **Nutzen** im Sinne einer Konkretisierung der Tätigkeit im Hinblick auf die Verwendung der Ergebnisse, z.B. „als Vorlage für die Entwicklung der Software“ oder „als Grundlage zur Abstimmung der Fachlichkeit mit dem Kunden“
- **Akteur** im Sinne einer Qualifikation oder einer über spezifische Zuständigkeiten definierten Projektrolle, z.B. „der Chef-Architekt“
- **Durchführungszeitpunkt** im Sinne einer Konkretisierung, in welchen typischen Projektphasen die Tätigkeit durchgeführt wird, z.B. „in einer Basiskonzeptphase vor Beginn der iterativen Entwicklung“
- **Ausprägung** im Sinne weiterer Details zur näheren Beschreibung der Tätigkeit, z.B. „grob auf Ebene der Hauptakteure“ oder „notiert mittels UML-Use Case-Diagrammen“.

Gerade die Zusatzinformationen zu Nutzen, Akteur, Durchführungszeitpunkt und Ausprägung ermöglichen erst die direkte Anwendbarkeit. Am Beispiel des Durchführungszeitpunktes wird das besonders deutlich: Entscheidend für die Effizienz ist oft nicht so sehr ob, sondern vielmehr wann eine bestimmte Tätigkeit durchgeführt wird. So besteht der Kern der gesamten agilen Methodik beispielsweise darin, nur wenig Planung und Konzeption vorab, d.h. vor Beginn der eigentlichen Umsetzungsarbeit im

Projekt durchzuführen. Vielmehr verteilen sich planerische und konzeptionelle Tätigkeiten stark auf die verschiedenen Iterationen der Entwicklung, um Changes besser aufnehmen und aus bereits Umgesetztem direkter lernen zu können. Zur Erfassung dieser Durchführungszeitpunkte legen wir ein allgemeines Raster zugrunde, das die relevanten Phasen und Meilensteine in einem Softwareprojekt enthält. Dieses ist in Abb. 6 dargestellt. Neben einer eindeutigen Zuordnung einer bestimmten Tätigkeit zu einer bestimmten Phase im Raster als BeST-Practice kann es übrigens natürlich auch eine BeST-Practice sein, bestimmte Tätigkeiten *nie* durchzuführen. Sehr viel Projekteffizienz entsteht, in bestimmten Projektkontexten unnötige Tätigkeiten einfach wegzulassen.

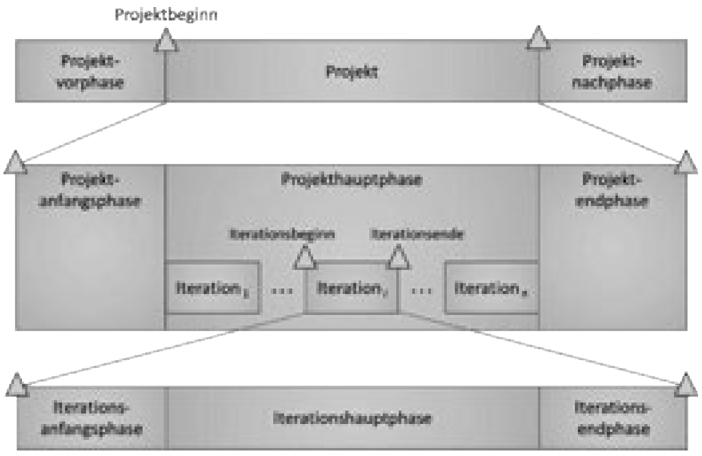


Abbildung 6: Projektphasenraster

Damit ist eine BeST-Practice strukturell hinreichend beschrieben. Nun bringen wir Projektkontext und strukturierte BeST-Practices zu *Effizienzaussagen* zusammen. Dazu weisen wir einer dedizierten Menge von ganz konkret formulierten Einflussfaktoren *E* eine BeST-Practice in Form von Aussagen *A* zu Tätigkeit, Nutzen, Akteur, Durchführungszeitpunkt und Ausprägung zu und erklären dies als „effizient“ im Sinne eines günstigen Verhältnisses des Grades der Abdeckung der Anforderungen (messbar z.B. in Form der Zufriedenheit des Kunden) zum dafür zu erbringenden personellen Aufwand (messbar z.B. in Personentagen). Abb. 7 illustriert diesen Ansatz.

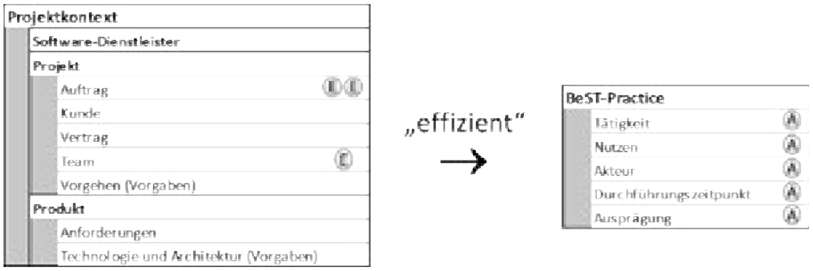


Abbildung 7: Effizienzaussage

Im unten beschriebenen Forschungsprojekt erarbeiten wir hierfür auch eine formale Darstellung, die konkrete Effizienzgrade zuordnet und so auch „Worst“-Practices beschreiben kann. Dazu führen wir hier aber keine weiteren Details ein, weil der Schwerpunkt dieses Beitrags auf dem generellen Prinzip liegen soll.

Eine solche Effizienzaussage entspricht nun genau den oben bereits erwähnten kontextabhängigen BeST-Practices. Dabei können aufgrund des beschriebenen Ansatzes der Ganzheitlichkeit einer Beschleunigten Softwaretechnik die in dieser Form konkretisierten Tätigkeiten bzw. Methoden *allen* fachlichen Domänen angehören. Dieser Betrachtungswinkel ist ebenfalls relativ neu, denn nicht nur werden die oben bereits referenzierten kontextgetriebenen Ansätze wie [WK92], [HE10], [EK11] oder [CO12] nur selten konkret bei den Best Practices, sie konzentrieren sich zusätzlich ausschließlich auf Auswahl und Zuschnitt des Vorgehensmodells im Sinne der Managementmethodik und lassen anderen Domänen wie Analyse-, Architektur- oder Entwicklungsmethodik außen vor. Unser Ansatz bezieht hingegen bewusst alle Aufgabenbereiche der Softwaretechnik mit ein und so ist auch das im Folgenden näher beschriebene Forschungsprojekt aufgesetzt.

5. Forschungsprojekt

In Zusammenarbeit mit der Hochschule Darmstadt haben wir in diesem Jahr ein Forschungsprojekt gestartet, indem es darum geht, kontextabhängige BeST-Practices zu identifizieren. Da dies nicht im ersten Schritt flächendeckend für Aufgabenbereiche aus allen fachlichen Domänen erfolgen kann, haben wir drei dedizierte Aufgabenbereiche für diese Betrachtung ausgewählt:

- **Projektplanung** und Festlegung des Managementmodells (Domäne Projektmanagement): Das ist der Bereich, der auch in anderen Arbeiten bereits nach kontextspezifischen Aussagen zum Vorgehen untersucht wurde. Hier geht es uns speziell darum aufzuzeigen, dass sich die Aussagen zum Vorgehen in Form der kontextspezifischen BeST-Practices deutlich konkreter formulieren lassen.
- **Aufwandsbetrachtung** als Aufgabenbereich, der Unteraufgaben wie Aufwandsschätzung, Aufwandsberechnung und Aufwandscontrolling umfasst (ebenfalls Domäne Projektmanagement): Hier geht es uns insbesondere darum zu zeigen, dass der Projektkontext nicht nur innerhalb der Methoden berücksichtigt werden kann, sondern dass insbesondere bei Auswahl und Zuschnitt der Methoden der Projektkontext eine wesentliche Rolle spielt.
- **Systemarchitekturdefinition** (Domäne Architektur): Hier geht es insbesondere darum aufzuzeigen, dass auch für Nicht-Projektmanagement-Domänen die projektkontextspezifische Auswahl von Methoden und die Unterstützung durch kontextspezifische BeST-Practices von Nutzen ist. Die Definition der Architektur eines Softwaresystems erfolgt selbstverständlich ausgehend von den (insbesondere nicht-funktionalen) Anforderungen. Diese sind projektspezifisch und somit Teil des Projektkontextes (vgl. Abb. 4). Es gibt aber darüber hinaus weitere Einflussfaktoren, die einen erheblichen Einfluss auf die Architektur haben.

Für diese drei Aufgabenbereiche werden jeweils ausgewiesene Domänenexperten mit einschlägiger Projekterfahrung in Form einer qualitativen Umfrage (angelehnt an Methoden der Sozialwissenschaften, z.B. [BD06]) mittels Interviews befragt. Die Ergebnisse werden aufbereitet und in einer zweiten Runde mittels strukturierter Fragebögen von einer weiteren Gruppe von Experten und durch einen Abgleich mit der Literatur verifiziert.

Die Umfangsbeschränkung für diesen Beitrag lässt es weder zu, das Vorgehen detailliert zu erläutern, noch alle Ergebnisse im Überblick darzustellen. Wir beschränken und daher vielmehr darauf, für jeden der drei Bereiche jeweils *eine* kontextspezifische BeST-Practice vorzustellen, die noch nicht allgemein bekannt ist. ^{E} bezeichnet dabei Einflussfaktoren, ^{A} bezeichnet Aussagen zur BeST-Practice.

Beispiel Projektplanung: Eine von vielen wichtigen Fragestellungen bei der Planung eines Projektes ist es, zu entscheiden, wer im Entwicklungsteam welche Aufgaben übernimmt. Dies kann „klassisch“ durch den Projektleiter oder „agil“ durch das selbstorganisierte Team erfolgen. Eine Frage, die sich dabei aber häufig stellt, ist in beiden Fällen die Gleiche: „Wie schneide ich meine Aufgaben sinnvoll?“. Angenommen, die Entwicklung erfolgt auf Basis einer Vorgabe zur Softwarearchitektur, in der vertikal *fachliche Komponenten* (entsprechend anwendungsfachlicher Domänen oder Service-Gruppen) und horizontal *technische Schichten* (z.B. Präsentations-, Dialogkern-, Servicezugriffs-, Geschäftslogik-, Datenzugriffs-, und Datenschicht) zu definieren sind (z.B. gemäß [Si04]). Welche Form der Verteilung der Arbeit auf die Team-Mitglieder ist dann die effizienteste?

Ergebnis der Erhebung ist, dass das nicht allgemeingültig beantwortet werden kann, sondern insbesondere vom Umfang der Fachlichkeit abhängt. Für größere Projekte^E mit umfangreicher Fachlichkeit^E sollte die Verteilung der Aufgaben^A vertikal^A d.h. entlang der fachlichen Komponenten erfolgen, vorausgesetzt das Team ist technologisch hinreichend kompetent^E, sich in die unterschiedlichen Technologien der technischen Schichten angemessen schnell einzuarbeiten. Das fachliche Know-how dominiert hier also die Projekteffizienz. Dieser Zusammenhang ist noch reletiv bekannt. Anders ist das bei kleineren Projekten^E mit geringer fachlicher Komplexität^E. Hier ist allerdings nicht die horizontale Verteilung der Aufgaben auf die Entwickler zu bevorzugen, sondern vielmehr eine Vermeidung rein vertikaler oder horizontaler Aufgabenzuordnung. Tatsächlich ist das Team in einem solchen Projektkontext in Summe am effizientesten, wenn im Rahmen der Selbstorganisation^A eine möglichst optimale Verteilung von Aufgaben^A über die fachlichen Komponenten und die technischen Schichten hinweg^A erfolgen kann. Grund dafür ist die gerade für kleine Projekte wichtige optimale Verteilung von Wissen im Team im Sinne eines angemessenen Generalistentums.

Beispiel Aufwandsbetrachtung: Heutzutage werden viele Softwareprojekte noch immer klassisch bottom-up von Experten explizit geschätzt, die letztlich subjektiv erfolgt. Dabei wäre eine auf Zahlen beruhende top-down Methode wie die UCP-Methode, bei der im Idealfall die fachliche Komplexität in Use Case Points explizit vermessen wird (objektiv entlang klarer Richtlinien und spezifisch für den jeweiligen Systemtyp) und auch der Produktivitätsfaktor des Teams aus bereits durchgeführten

Projekten abgeleitet werden kann, aufgrund seiner Objektivität eigentlich zu bevorzugen. Aber selbst wenn der Idealfall eintritt, d.h. eine solche top-down Methode bei „Beherrschung“ der Zahlen prinzipiell angewendet werden könnte, gibt es doch Einflussfaktoren, die dies verhindern können.

So ist als Ergebnis der Erhebung das Verhältnis der UCPs, die in einem bestimmten Zeitraum implementiert werden, zur Teamkapazität in Full-time equivalents (FTEs), die diese implementieren sollen, entscheidend für die Methodenentscheidung. Ist das Verhältnis der Anzahl der UCPs zur Anzahl der FTEs sehr klein^E, sollte auf jeden Fall eine Expertenschätzung^A erfolgen, da die gemessenen Produktivitätsfaktoren in diesen Fällen nicht mehr aussagekräftig sind. Das liegt daran, dass im Fall eines pathologischen Verhältnisses von Mitarbeiterkapazität zu Projektlaufzeit die Effizienz exponentiell abnehmen kann. Im anderen Fall ist hingegen eine top-down Methode^A zu bevorzugen.

Beispiel Systemarchitekturdefinition: Bereits in [Vo12a/b] haben wir dargelegt, dass die Arbeit mit Referenzarchitekturen, die zum zu entwickelnden Systemtyp passen, sehr effizient sein kann, allerdings nur, wenn diese Referenzarchitekturen auch auf die (vor allem nicht-funktionalen) Anforderungen zugeschnitten werden. Für die konkrete Durchführung der Tätigkeit des Zuschnitts einer Referenzarchitektur gibt es aber auch Abhängigkeiten zu anderen Einflussfaktoren des Projektkontexts.

So hat die Erhebung ergeben, dass ein Zuschnitt^A mit Tendenz zur Vereinfachung^A in einem Envisioning in der Projektanfangsphase^A zu einem effizienten Vorgehen führt, wenn die initial und auch in Zukunft mit der Entwicklung betrauten Mitarbeiter über ein hohes Wissen im Bereich Softwarearchitektur verfügen^E und die Entwicklung nicht über verteilte Standorte^E erfolgt.

6. Zusammenfassung und Ausblick

Wir haben gezeigt, dass BeST-Practices und insbesondere auch kontextabhängige BeST-Practices ein wichtiges Mittel sind, um die Effizienz in Softwareprojekten durch sehr konkrete Vorgehenshinweise zu unterstützen, die etablierte Methoden nicht ersetzen sondern ergänzen und besser nutzbar machen. Mit der Beschleunigten Softwaretechnik, die inhaltlich stark auf diese praxiserprobten BeST-Practices setzt, bearbeiten wir ein Feld, das weiter an Bedeutung gewinnen wird.

Die Definition des BeST-Frameworks und der allgemeingültigen wie auch kontextabhängigen BeST-Practices ist inzwischen soweit fortgeschritten, dass wir Beschleunigte Softwaretechnik in Form einer zweitägigen BeST-Foundation bereits erfolgreich schulen. Neben der inhaltlichen Weiterentwicklung der Beschleunigten Softwaretechnik (z.B. in Form weiterer Forschungsprojekte wie oben beschrieben) und der Diskussionen innerhalb der Community (z.B. in Form dieses Papers) steht auch die weitere Verbreitung des Wissens im Vordergrund. So sind wir mit mehreren Hochschulen im Gespräch zu Vorlesungsbeiträgen oder ganzen Vertiefungsvorlesungen zur Beschleunigten Softwaretechnik.

Literaturverzeichnis

[BD06] Bortz, J., Döring, N.: Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler. Springer, 2006

[Bo81] Boehm, B.: Software Engineering Economics. Prentice-Hall, 1981

[Br75] Brooks, F.: The mythical man-month. Addison-Wesley, 1975

[CO12] Clarke, R., O'Connor, R.: The situational factors that affect the software development process: Towards a comprehensive reference framework. In Information and Software Technology, Volume 54, S. 433–447, 2012

[DL10] Dede, B., Lioufko, I.: Situational Factors Affecting Software Development Process Selection. Thesis, Universität Gothenburg, 2010

[EH08] Engels, G., Hess, A., Humm, B., Juwig, O., Lohmann, M., Richter, J.-P., Voß, M., Willkomm, J.: Quasar Enterprise - Anwendungslandschaften serviceorientiert gestalten. dpunkt Verlag, 2008

[EK11] Engels, G., Kremer, M.: Situational Software Engineering: Ein Rahmenwerk für eine situationsgerechte Auswahl von Entwicklungsmethoden und Vorgehensmodellen. In Heiß, H., Pepper, P., Schlinghoff, H., Schneider, J. (Hrsg.): Informatik 2011 – Informatik schafft Communities, Berlin; Lecture Notes in Informatics (LNI), Volume P-192, S. 469, Gesellschaft für Informatik, Bonn, 2011

[Fr09] Frohnhoff, S.: Use Case Points 3.0 : Implementierung einer Use Case bezogenen Schätzmethode für das Software-Engineering betrieblicher Informationssysteme. Dissertation, Universität Paderborn, 2009

[Fr12] Fryba, M.: Schäden in Milliarden-Höhe durch mangelhafte Softwareentwicklung. IT-Business, April 2012 (http://www.it-business.de/software/weiteres/articles/361042/)

[HE10] Heinemann, M., Engels, G.: Auswahl projektspezifischer Vorgehensstrategien. In Linssen, O. Greb, T., Kuhrmann, M., Lange, D., Höhn, R. (Hrsg.): Integration von Vorgehensmodellen und Projektmanagement, S. 132–142, Shaker Verlag, 2010

[LL10] Ludewig, J., Lichter, H.: Software Engineering: Grundlagen, Menschen Prozesse, Techniken. 2. Auflage, dpunkt Verlag, 2010

[PMI08] Project Management Institute (Hrsg.): A Guide to the Project Management Body of Knowledge. Project Management Institute, 2008

[SBV04] Schalken, J., Brinkkemper, S., van Vliet, H.: Discovering the Relation between Project Factors and Project Success in Post-mortem Evaluations. In Proceedings of the 11th European Software Process improvement conference (EUROSPI 2004), Volume 3281, Lecture Notes in Computer Science, Springer, 2004

[Sie04] Siedersleben, J.: Moderne Softwarearchitektur: umsichtig planen und robust bauen mit Quasar. dpunkt.Verlag, 2004

[Vo12a] Voß, M.: Softwarearchitektur in agilen Projekten – von Kathedralen und IT-Systemen. OBJEKTspektrum, Nr. 3/2012, S. 10-14, SIGS DATACOM, 2012

[Vo12b] Voß, M.: Architektur in einer Beschleunigten Softwaretechnik. Tagungsunterlagen der Architekturen 2012 - Jahrestagung der GI-Fachgruppe Software-Architektur und Objektorientierte Softwareentwicklung, Paderborn, Gesellschaft für Informatik, 2012 (http://is.uni-paderborn.de/architekturen2012/programm)

[VW10] van't Wout, J., Waage, M., Hartman, H., Stahlecker, M., Hofman, A.: The Integrated Architecture Framework explained. Springer, 2010

[WK92] Welke, R., Kumar, K.: Method engineering: a proposal for situation-specific methodology construction. In Cotterman, Senn (Hrsg.): Systems Analysis and Design: A Research Agenda, S. 257–268. Wiley, Chichester, 1992

SE | 13 SOFTWARE ENGINEERING

Workshops

3. Workshop zur Zukunft der Entwicklung softwareintensiver eingebetteter Systeme (ENVISON2020)

Ottmar Bender[1], Wolfgang Böhm[2], Stefan Henkler[3], Dietmar Sander[4], Andreas Vogelsang[2], Thorsten Weyer[5]

[1] EADS Deutschland GmbH
Geschäftsbereich Cassidian
Wörthstr. 85
89077 Ulm
ottmar.bender@cassidian.com

[2] Institut f. Informatik
Technische Universität München
Boltzmannstr. 3
85748 Garching b. München
boehmw@in.tum.de
vogelsan@in.tum.de

[3] OFFIS e.V.
Industriebereich Verkehr
Escherweg 2
26121 Oldenburg
stefan.henkler@offis.de

[4] Airbus Operations GmbH
Process, Methods & Tools
Kretslag 10
21129 Hamburg
dietmar.sander@airbus.com

[5] paluno – The Ruhr-Institute for Software Technology
Universität Duisburg-Essen
Gerlingstr. 16
45127 Essen
thorsten.weyer@paluno.uni-due.de

Heutzutage unterstützen softwareintensive eingebettete Systeme mehr oder weniger sichtbar den Menschen in vielen Bereichen des täglichen Lebens. Beispielsweise verbessern solche Systeme im Automobil die Sicherheit, regulieren das Klima in Gebäuden, unterstützen das Steuern von Flugrouten und die Flugstabilität in modernen Flugzeugen oder steuern medizinische Geräte bis hin zu ganzen Industrieanlagen. Experten prognostizieren für die Zukunft eine rasante Zunahme softwareintensiver, eingebetteter Systeme und deren Vernetzung in Systemverbünden. Diese Entwicklung wird in dramatischer Weise durch das entstehen umfassender „Cyber Physical Systems“ verstärkt, die in immer stärkeren Maße die transparente Integration von Softwaresystemen und der realen Welt forcieren.

Die kontinuierliche Ausweitung des Funktionsumfangs, steigende Qualitätsansprüche und die zunehmende Vernetzung softwareintensiver eingebetteter Systeme führen gleichzeitig zu einer signifikanten Zunahme des Umfangs und der Komplexität dieser Systeme und der resultierenden Systemverbünde. Um solche Systeme auch weiterhin effektiv und effizient entwickeln zu können und dabei planbare und nachvollziehbare Entwicklungsprozesse zu gewährleisten, müssen deren Umfang und Komplexität bereits im Engineering systematisch beherrscht werden. Existierende Techniken und Methoden stoßen schon heute (z.B. aufgrund des Zeit- und Kostendruck in Entwicklungsprojekten) häufig an ihre Grenzen. Dies hat unmittelbar zur Konsequenz, dass existierende Ansätze Techniken und Methoden aufgrund der wachsenden Herausforderungen in Frage gestellt und wenn notwendig aufgegeben bzw. in Teilen neu konzipiert werden müssen.

Der Workshop ENVISION 2020 verfolgt das Ziel, die Erarbeitung und Diskussion zukünftiger Konzepte, Techniken, Vorgehensweisen und Methoden zur Entwicklung softwareintensiver eingebetteter Systeme zu fördern. Ein besonderes Augenmerk gilt dabei durchgängigen modellbasierten Entwicklungsansätzen.

In drei Kategorien von Beiträgen werden im Rahmen des Workshops jeweils dedizierte Fragestellungen adressiert:

Kategorie 1: Forschungsherausforderungen

Kategorie 2: Lösungsansätze zur Bewältigung konkreter Herausforderungen

Kategorie 3: Lösungsansätze in der industriellen Praxis.

Der Workshop ENVISION2020 wird organisiert durch die Forschungsinitiative „Software Plattform Embedded Systems XT“ (SPES2020_XT), die sich zum Ziel gesetzt hat, den Stand der Wissenschaft und Praxis auf dem Gebiet des Engineering von softwareintensiven eingebetteten Systemen mit Hilfe durchgängiger modellbasierter Ansätze nachhaltig zu verbessern. SPES_XT wird gefördert durch das BMBF (Förderkennzeichen 01IS12005). Weitere Informationen finden sich unter: http://www.spes2020.de

Traceability – Nutzung und Nutzen

Workshop innerhalb der Konferenz SE2013

Elke Bouillon[1], Matthias Riebisch[2], Ilka Philippow[1]

[1]Fachgebiet Softwaresysteme/Prozessinformatik
Technische Universität Ilmenau
Postfach 100565, D-98684 Ilmenau
elke.bouillon@tu-ilmenau.de
ilka.Philippow@tu-ilmenau.de

[2]Fachbereich Informatik, Universität Hamburg
Vogt-Koelln-Str. 30, D-22527 Hamburg
riebisch@informatik.uni-hamburg.de

Auf dem Workshop sollen Forschungsergebnisse auf dem Gebiet der Traceability in Softwareprojekten vorgetragen und diskutiert werden. Traceability ist ein wichtiger Bestandteil des Softwareentwicklungsprozesses. Man versteht darunter die Nachvollziehbarkeit der Umsetzung von Anforderungen über die verschiedenen Entwicklungsstadien hinweg, von einzelnen Entwicklungsaktivitäten, sowie von bestehenden Abhängigkeiten und das über Änderungen während des gesamten Lebenszyklus' der Software hinweg. Der Einsatz von Traceability in der Softwareentwicklung bietet zahlreiche Vorteile wie steigende Qualität der entstehenden Software und eine Vereinfachung von Weiterentwicklungen. Trotzdem zeigen Studien, dass es beim Einsatz von Traceability in der Praxis nach wie vor Defizite gibt.

Der Hauptschwerpunkt des Workshops liegt bei Ansätzen, die zu einer Balance zwischen Aufwand und Nutzen von Traceability beitragen. Der Workshop besteht aus Vorträgen zur Vorstellung der Beiträge und bietet darüber hinaus breitem Raum für Diskussionen.

Schwerpunkte der Diskussion sind beispielsweise:

- Methoden und Werkzeuge zur Unterstützung von Traceability
- Nutzungsszenarien von Traceability
- Traceability in einem agilen Umfeld - sinnvoll und praktikabel?
- Studien zum Einsatz von Traceability

Ebenfalls von Interesse sind Erfahrungsberichte über den Einsatz von Traceability in der Praxis. Der Workshop soll aktuelle Forschungsergebnisse darstellen und diese mit Erfahrungen aus der Praxis zusammenführen. Der Workshop richtet sich daher ausdrücklich auch an Interessenten aus Wirtschaft und Industrie. Dadurch soll der Wissenstransfer gefördert und aktuelle Fragen für eine anwendungsnahe Forschung entwickelt werden.

Workshop on Managed Software Evolution

Ursula Goltz, Lukas Märtin

Technische Universität Braunschweig
Institut für Programmierung und Reaktive Systeme
Mühlenpfordtstr. 23
38106 Braunschweig
goltz@ips.cs.tu-bs.de
maertin@ips.cs.tu-bs.de

Abstract: Das DFG Schwerpunktprogramms 1593 **Design for Future - Managed Software Evolution** wurde gegründet, um fundamentale neue Ansätze für den Bereich langlebiger Software-Systeme zu entwickeln.

Die bisherige Forschung im Bereich Software Engineering konzentriert sich stark auf die Phase der initialen Software-Entwicklung. Ziel des Schwerpunktprogramms ist im Gegensatz dazu ein integrierter Ansatz, der die kontinuierliche Weiterentwicklung von Software-Systemen im Hinblick auf ständig wechselnde Anforderungen, neuentstehende Technologien, und die Integration neuer Software-, Hardware- und System-Komponenten unterstützt. Damit sollen unter anderem die bekannten Probleme der Legacy-Software und der Anpassung an neue Plattformen vermieden werden. Es werden neue Ansätze, Methoden und Werkzeuge zur Entwicklung von "jungbleibender" Software, die ihre ursprüngliche Funktionalität und Qualität behält und sich kontinuierlich während der gesamten Lebensdauer verbessert, entwickelt. Wir stellen ein neues Paradigma auf, indem einerseits Entwicklung, Anpassung und Evolution von Software und deren Plattformen und andererseits Betrieb, Überwachung und Wartung nicht mehr getrennt, sondern integriert betrachtet werden. Durch ausgewählte Fallstudien werden insbesondere zwei konkrete Anwendungsbereiche betrachtet: Informationssysteme und Produktionssysteme in der Automatisierungstechnik.

Zwei eingeladene Vorträge aus Wissenschaft und Industrie führen in die Thematik des Schwerpunktprogramms ein und geben einen Überblick über relevante Forschungsfelder.

Durch Vorträge aus dem Schwerpunktprogramm und eine ausführliche Poster-Session werden die Konzepte der 13 Teilprojekte des Schwerpunktprogramms erstmalig öffentlich präsentiert und deren Beitrag zur Vision des Schwerpunktprogramms diskutiert. Die beiden wichtigsten Themenbereiche für die erste Phase des Schwerpunktprogramms werden dargestellt:

Knowledge Carrying Software: Geeignete Meta-Modelle erlauben bei der Evolution die Erhaltung und den Zugriff auf Wissen, das während der Systementwicklungsphasen gewonnen wird.

Methoden und Prozesse: Innovative Methoden und Vorgehensmodelle zur umfassenden Unterstützung der Integration der Software-Entwicklung und Evolution während des gesamten Software-Lebenszyklus müssen bereitgestellt werden.

In zwei weiteren Vorträgen werden die beiden SPP-übergreifenden Fallstudien mit geeigneten Evolutionsszenarien diskutiert.

Ziel des Workshops ist neben dem Ausbau der Vernetzung zwischen den Teilnehmern des Schwerpunktprogramms vor allem die intensive Diskussion der Konzepte mit Interessierten aus Wissenschaft und Industrie.

In einer separaten Session werden Kurz-Tutorien zu den im SPP verwendeten Plattformen, Methoden und Werkzeugen angeboten, die für den Austausch im Schwerpunktprogramm wichtig und gleichzeitig für Außenstehende sehr interessant sind.

ZeMoSS-Workshop: Zertifizierung und modellgetriebene Entwicklung sicherer Software

Michaela Huhn[1] Stefan Gerken[2] Carsten Rudolph[3]

[1] Institut f¨ur Informatik, Technische Universit¨at Clausthal
[2] IC MOL RA R&D, Siemens AG, Braunschweig
[3] Fraunhofer Institut f¨ur Sichere Informationstechnologie (SIT), Darmstadt

Software ist ein wesentlicher Innovationsfaktor bei vielen technischen Produkten und Infrastrukturen.
Mit dem vielfältigen Einsatz softwaregesteuerter Systeme wachsen die Qualitätsanforderungen, insbesondere in den Bereichen funktionale Sicherheit und Informationssicherheit. In der Luft- und Raumfahrt, der Energieerzeugung und im Schienenverkehr, aber auch in der Medizintechnik wie auch in den Bereichen Automobiltechnik und mobile Systeme sind die Zertifizierung und der Nachweis der Sicherheit kritischer Systeme und softwarespezifische Sicherheitsnormen international etabliert und bindend. Mehrere domänenübergreifende Herausforderungen bei der Entwicklung sicherer Software werden im Workshop adressiert:

- Modellgetriebene Software-Entwicklung wird in der Industrie immer wichtiger und für höhere Sicherheitsanforderungsstufen seit langem in Sicherheitsnormen als dringend empfohlen klassifiziert. Normen können aber nur unzureichend auf den Stand von Wissenschaft und Technik eingehen. Neue Methoden und Werkzeuge müssen im Zertifizierungsprozess akzeptiert werden, auch wenn zu ihnen keine normativen Aussagen vorliegen. Somit steht einer Qualitätserhöhung von Software in sicherheitsrelevanten Anwendungen durch neue Methoden immer die fehlende normative Erwähnung dieser Methode gegenüber.

- Durch die zunehmende Vernetzung innerhalb kritischer Infrastrukturen und die Anbindung
  mobiler Endgeräte werden viele Anwendungen erst ermöglicht, aber es entstehen auch neue Risiken aus der wechselseitigen Abhängigkeit von Informationssicherheit und funktionaler Sicherheit. Hier sind neue Methoden gefragt, die eine integrierte Modellierung von Safety und Security in der Risikoanalyse, der Entwicklung und dem Nachweis unterstützen.

Der ZeMoSS-Workshop soll den Austausch über offene Punkte und Lösungsansätze zu zentralen Herausforderungen in der Software-Entwicklung für sicherheitskritische Systeme und Infrastrukturen domänenübergreifend zwischen Teilnehmern aus Industrie und Forschung fördern.

6. Arbeitstagung Programmiersprachen (ATPS 2013)

Jens Knoop[1], Janis Voigtländer[2]

[1] Institut für Computersprachen, TU Wien Argentinierstr. 8 / E185.1, A-1040 Wien, Österreich knoop@complang.tuwien.ac.at [2] Institut für Informatik, Universität Bonn Römerstraße 164, D-53117 Bonn, Deutschland
jv@iai.uni-bonn.de

Die Tagung dient dem Austausch zwischen Forschern, Entwicklern und Anwendern, die sich mit Themen aus dem Bereich der Programmiersprachen beschäftigen. Alle Programmierparadigmen sind 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. Typische, aber nicht ausschließliche Themenbereiche sind:

- Entwurf von Programmiersprachen und anwendungsspezifischen Sprachen
- Implementierungs- und Optimierungstechniken
- Analyse und Transformation von Programmen
- Ressourcenanalyse (Zeit, Speicher, Leistungsverbrauch)
- Typsysteme
- Semantik und Spezifikationstechniken
- Modellierungssprachen, Objektorientierung
- Intra- und Internet-Programmierung
- Programm- und Implementierungsverifikation
- Werkzeuge und Programmierumgebungen
- Frameworks, Architekturen, generative Ansätze
- Erfahrungen bei exemplarischen Anwendungen
- Verbindung von Sprachen, Architekturen, Prozessoren

Ebenfalls von Interesse sind Arbeiten zu Techniken, Methoden, Konzepten oder Werkzeugen, mit denen Sicherheit und Zuverlässigkeit bei der Ausführung von Programmen erhöht werden können. Die Tagung richtet sich ausdrücklich auch an Interessenten aus Wirtschaft und Industrie.

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

First European Workshop on Mobile Engineering (ME’13)

Walid Maalej[1], Dennis Pagano[2], Bernd Bruegge[2]

[1]Universität Hamburg
Vogt-Kölln-Straße 30, D-22527 Hamburg
maalej@informatik.uni-hamburg.de

[2]Technische Universität München
Boltzmannstraße 3, 85748 Garching
pagano@cs.tum.edu
bruegge@cs.tum.edu

Abstract: ME’13 focuses on potentials and challenges of mobile computing for the software engineering community. The workshop discusses emerging ideas, methodologies, frameworks, tools, as well as industrial experiences with the engineering and management of mobile services and applications, and aims at establishing a research community around these topics. Furthermore, the workshop provides an interactive exchange platform between the software engineering community and industrial practitioners in the mobile computing area.

1. Motivation

Mobile is becoming mainstream. Mobile devices are among the most sold computers in the world. For instance, Apple sold over 70 million iPhone 4S only in 2011; Samsung recently announced over 30 million sold Galaxy S III only in 2012. One of the reasons behind the mobile hype is the ever-growing power of mobile devices, which often outperform a typical five-year-old desktop computer. Moreover, mobile devices offer novel human-computer interfaces like touch-screens or speech recognition, and employ powerful sensors, such as GPS, gyroscopes, or video cameras. These interfaces and sensors enable a fully new spectrum of context-aware, personalized, and intelligent software services and applications.

The powerful, modern software frameworks and libraries, which enable the design of new mobile “apps” in several hours, together with the huge, highly dynamic user communities make mobile platforms very attractive also for developers. As of September 2012, over 700,000 applications are available in the Apple AppStore, more than 500,000 in Google Play[1]. Download numbers are astronomic – around 1 billion per month in the Apple AppStore – thanks to the application distribution platforms, where users can buy and deploy apps with just one click.

Mobile brings new potentials and challenges for the software engineering community. As researchers and practitioners we need to ask ourselves whether common software

[1] http://www.appbrain.com/stats/

engineering tools, methods, and processes are appropriate for designing and maintaining mobile software services and applications, whether mobility has an impact on how software needs to be designed, and whether there are any special circumstances at all for our community – or whether mobile is only "yet another platform". This workshop discusses issues, approaches, and tools regarding the engineering of mobile software applications and services.

2. Workshop Objective and Topics

The workshop objective is twofold. First, it aims at establishing a community around potentials and challenges provided by mobile services for software engineering research and practice. Second, it strives to identify open issues, novel approaches, and future research directions in the area of software engineering for mobile applications and services.

The topics of the workshop include, but are not restricted to:

- Requirements engineering for mobile services, in particular:
    - User communities, user involvement, and user feedback
    - Context aware mobile services
    - Usability, privacy, security, and performance
- Design and implementation of mobile services, in particular:
    - Tools, frameworks, and design patterns for mobile applications and services
    - Virtualization approaches
- Development environments and frameworks for mobile services and applications
- Mobile service deployment
    - App engines, app stores
    - Integration of mobile apps with conventional software systems
    - Beta testing platforms
- Case studies, success and failure stories on engineering mobile services
- Using mobile services by software engineering teams
- Mobile computing in software engineering education

3. Workshop Organization

Workshop Organizers:
- Walid Maalej, University of Hamburg, Germany
- Dennis Pagano, Technische Universität München, Germany
- Bernd Bruegge, Technische Universität München, Germany

Program Committee:

- Raian Ali, Bournemouth University, UK
- Arosha Bandara, Open University, UK
- Roberto Bertoldi, WIND, Italy
- Patrick Blitz, Weptun, Germany
- Tilo Boehmann, University of Hamburg, Germany
- Miguel Juan, S2, Spain
- Jan Marco Leimeister, University of Kassel, Germany
- Inah Omoronyia, University of Glasgow, Scotland
- Martin Ott, Equinux AG, Germany
- Liliana Pasquale, Lero, Ireland
- Kurt Schneider, University of Hannover, Germany
- Norbert Seyff, University of Zurich, Switzerland

International Workshop on Comparison and Versioning of Software Models (CVSM 2013)

Pit Pietsch[1+], Udo Kelter[1] and Jan Oliver Ringert[2‡]

[1] Universität Siegen
Hölderlinstraße 3, 507076 Siegen, Germany
pietsch, kelter@informatik.uni-siegen.de
[2] Software Engineering RWTH Aachen University
Ahornstraße 55, 52074 Aachen, Germany
ringert@se.rwth-aachen.de

The International Workshop Series on Comparison and Versioning of Models brings together scientists and practitioners in the field of model versioning. Particularly technologies like comparison, versioning, merging, patching and weaving of models are addressed. These technologies are indispensable to support model-based development methods and consequently they have been subject of intensive research in the last decade. A significant number of algorithms and tools have been developed. Empirical evaluations on these tools have been conducted so far mostly by suppliers of the technologies, typically using a small set of use cases and data sets. These evaluations cannot be reproduced or repeated with competing approaches due to the lack of available materials. Naturally, this hinders progress because no objective judgment of the different approaches can be made.

Hence, this year's issue of the CVSM is devoted to overcome this weakness in the state-of-the-art. The goal is to agree as the community as a whole on an initial set of standardized benchmarks. This set will enable tool providers to evaluate the quality of their tools on a common base and allows a comparison of competing approaches.

To reach this goal, we accept three types of submissions to this workshop: (1) *Performance Benchmarks*, usually consisting of several large models which are used to measure the runtime of different algorithms. (2) *Challenges*, which are usually small, artificially created models used to highlight certain quality aspects. (3) *Real Use Cases*, which help to assess and compare the discussed advantages and disadvantages of current algorithms in the context of real world application scenarios.

Every interested member of the community is invited to submit examples for inclusion in the benchmark set. All submissions will be discussed and evaluated by all participants of the workshop. Which examples will eventually be included in the benchmark set is decided during the workshop. It is planned to reevaluate and enhance the benchmark set in future issues of the CVSM workshop.

[+] P. Pietsch is supported by the DFG under grant KE499/5-1 QuDiMo.

[‡] J.O. Ringert is supported by the DFG under grant GK/1298 AlgoSyn.

5. Workshop „Design For Future – Langlebige Softwaresysteme“

Stefan Sauer[1], Benjamin Klatt[2], Thomas Ruhroth[3]

[1]Universität Paderborn, s-lab – Software Quality Lab
Zukunftsmeile 1, 33102 Paderborn
sauer@s-lab.upb.de
[2]FZI Forschungszentrum Informatik an der Universität Karlsruhe
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe
klatt@fzi.de
[3]TU Dortmund, Fakultät für Informatik
Otto-Hahn-Straße 14, 44221 Dortmund
thomas.ruhroth@cs.tu-dortmund.de
http://akl2s2.ipd.kit.edu/veranstaltungen/dff2013/

Auch Software altert! Dieses Problem ist vor allem bei großen betrieblichen Informationssystemen unter dem Begriff Legacy bekannt und wird sich in Zukunft noch weiter verschärfen. Zum einen gewinnen eingebettete Systeme immer größere Bedeutung, in denen aufwändige Software in langlebigen technischen Geräten eingesetzt wird. Zum anderen macht die steigende Vernetzung von Systemen in großen Anwendungslandschaften die Situation zunehmend komplexer. Diese Probleme haben enorme ökonomische Bedeutung. Wissenschaft und Industrie sind gefordert, neue Methoden der Softwaretechnik zu entwickeln, um die erheblichen Investitionen in große Softwaresysteme zu schützen und massive Probleme durch steigende Software-Erosion zu verhindern.

Aktuelle Ansätze der Softwaretechnik, vor allem modellbasierte Entwicklungsmethoden, Lifecycle-Management, Softwarearchitektur, Requirements und Re-Engineering, können die Situation verbessern, wenn sie geeignet weiterentwickelt und angewandt werden.

Der Arbeitskreis „Langlebige Softwaresysteme“ (AK L2S2) der GI-Fachgruppen Architekturen und Software-Reengineering will Wissenschaftler und Praktiker zusammenzubringen, die an diesen Themenstellungen Interesse haben. Im 5. Workshop „Design For Future – Langlebige Softwaresysteme“ sollen die oben geschilderte Entwicklung, Erfahrungen hierzu sowie Lösungsansätze sowohl aus praktischer als auch aus wissenschaftlicher Sicht beleuchtet werden, um die verschiedenen Facetten und Herausforderungen der Software-Alterung zu beherrschen. Es sollen sowohl Lösungen als auch praktische Erfahrungen betrachtet und diskutiert werden, um die Entstehung neuer Legacy-Probleme und die Erosion von Software zu verhindern. Beiträge werden insbesondere zu den folgenden Themen erwartet: anpassungsfähige und zukunftssichere Software-Architekturen; Evolution und Co-Evolution von Modellen und Code; Verhinderung von Software-Erosion; Re-Engineering zum Erkennen und Beheben von Legacy-Problemen; Entwicklungsmethoden und Lifecycle-Management für langlebige Softwaresysteme; Qualitätsmanagement für langlebige Softwaresysteme; Fallstudien zu den vorgenannten Themen; Praxis- und Erfahrungsberichte zu den vorgenannten Themen.