Gesellschaft für Informatik (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into the fields of
• Seminars
• Proceedings
• Dissertations
• Thematics
current topics are dealt with from the fields of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure the high level of the contributions.

The volumes are published in German or English.

Information: http://www.gi-ev.de/service/publikationen/lni/

ISSN 1617-5468
ISBN 978-3-88579-253-6

This volume contains papers from the Software Engineering 2010 conference held in Paderborn from February 22nd to 26th 2010. The topics covered in the papers range from software requirements, technologies or development strategies to reports that discuss industrial project experience.

Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.): Software Engineering 2010

159

GI-Edition

Lecture Notes in Informatics

Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)

Software Engineering 2010

22.–26. Februar 2010
Paderborn

Proceedings

Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)

Software Engineering 2010

Fachtagung des GI-Fachbereichs Softwaretechnik

22.-26.02.2010
in Paderborn

Gesellschaft für Informatik e.V. (GI)

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

Volume P-159

ISBN 978-3-88579-253-6
ISSN 1617-5468

Volume Editors
Gregor Engels
Markus Luckey
Wilhelm Schäfer
Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Warburger Str. 100
33098 Paderborn
Email: {engels,luckey,wilhelm}@upb.de

printed by Köllen Druck+Verlag GmbH, Bonn

Willkommen zur SE 2010 in Paderborn!

Die Tagung Software Engineering 2010 (SE 2010) ist die sechste Veranstaltung einer inzwischen etablierten Reihe von Fachtagungen, deren Ziel die Zusammenführung und Stärkung der deutschsprachigen Softwaretechnik ist. Die SE 2010 bietet ein Forum zum intensiven Austausch über praktische Erfahrungen, wissenschaftliche Erkenntnisse sowie zukünftige Herausforderungen bei der Entwicklung von Softwareprodukten bzw. Software-intensiven Systemen. Sie richtet sich gleichermaßen an Teilnehmer aus Industrie und Wissenschaft.

Die Software Engineering-Tagungsreihe wird vom Fachbereich Softwaretechnik der Gesellschaft für Informatik e.V. getragen. Die Software Engineering 2010 wird vom Lehrstuhl für Datenbank- und Informationssysteme sowie vom s-lab (Software Quality Lab) der Universität Paderborn veranstaltet.

Die SE 2010 bietet im Hauptprogramm begutachtete Forschungsarbeiten und eingeladene wissenschaftliche Vorträge. Von den 47 Einreichungen für das technisch-wissenschaftliche Programm wurden 17 Beiträge akzeptiert. Darüber hinaus werden in begutachteten und eingeladenen Praxisvorträgen am Industrietag aktuelle Problemstellungen, Lösungsansätze und gewonnene Erfahrungen präsentiert und zur Diskussion gestellt. Abgerundet wird das Programm durch SE FIT, ein Forum für Informatik-Transferinstitute, und ein Doktorandensymposium. Vor dem Hauptprogramm der Konferenz finden 11 Workshops sowie sechs Tutorials zu aktuellen, innovativen und praxisrelevanten Themen im Software Engineering statt.

Die Durchführung der Tagung Software Engineering 2010 wäre ohne die Mitwirkung vieler engagierter Personen nicht möglich gewesen. Ich bedanke mich besonders bei meinen Kollegen Wilhelm Schäfer für die Planung des Industrietags, Ralf Reussner für die Koordination des Workshop- und Tutorialprogramms, Alexander Pretschner für die Organisation des Doktorandensymposiums und Stefan Sauer für die Planung und Durchführung des SE FIT. Ganz besonders bedanke ich mich bei meinem Mitarbeiter Markus Luckey für seinen unermüdlichen Einsatz rund um die Organisation der Tagung, sowie bei meiner Sekretärin Beatrix Wiechers, meinem Techniker Friedhelm Wegener und bei allen Mitgliedern meiner Forschungsgruppe für die große Unterstützung.

Paderborn, im Februar 2010

Gregor Engels

Tagungsleitung
Gregor Engels, Universität Paderborn

Leitung Industrietag
Wilhelm Schäfer, Universität Paderborn

Leitung Workshops und Tutorials
Ralf Reussner, Karlsruher Institut für Technologie

Tagungsorganisation
Markus Luckey, Universität Paderborn
Friedhelm Wegener, Universität Paderborn
Beatrix Wiechers, Universität Paderborn

Programmkomitee
Klaus Beetz, Siemens AG
Manfred Broy, TU München
Bernd Brügge, TU München
Jürgen Belz, Hella KGaA Hueck & Co.
Jürgen Ebert, Universität Koblenz-Landau
Martin Glinz, Universität Zürich
Michael Goedicke, Universität Duisburg-Essen
Klaus Grimm, Daimler AG
Volker Gruhn, Universität Leipzig
Wilhelm Hasselbring, Christian-Albrechts-Universität zu Kiel
Stefan Jähnichen, TU Berlin
Matthias Jarke, RWTH Aachen
Gerti Kappel, TU Wien
Udo Kelter, Universität Siegen
Roger Kilian-Kehr, SAP AG
Claus Lewerentz, BTU Cottbus
Horst Lichter, RWTH Aachen
Peter Liggesmeyer, TU Kaiserslautern
Oliver Mäckel, Siemens AG
Florian Matthes, TU München
Barbara Paech, Universität Heidelberg
Klaus Pohl, Universität Duisburg-Essen
Alexander Pretschner, TU Kaiserslautern
Andreas Rausch, TU Clausthal
Ralf Reussner, Karlsruher Institut für Technologie
Bernhard Rumpe, RWTH Aachen
Eric Sax, MBtech Group
Wilhelm Schäfer, Universität Paderborn
Andy Schürr, TU Darmstadt
Rainer Singvogel, msg systems AG
Markus Voß, Capgemini sd&m AG
Andreas Winter, Universität Oldenburg

Mario Winter, Fachhochschule Köln
Heinz Züllighoven, Universität Hamburg
Albert Zündorf, Universität Kassel

Weitere Gutachter

Arne Beckhaus
Christian Berger
Christian Bimmermann
Fabian Christ
Stephen Dawson
Florian Deißenböck
Alexander Delater
Markus von Detten
Thomas Flor
Andreas Gehlert
Christian Gerth
Alexander Gruler
Matthias Heinrich
Steffen Helke
Stefan Henkler
Jens Herrmann
Jörg Holtmann
Ruben Jubeh
Elmar Jürgens
Lars Karg
Timo Kehrer
Moritz Kleine
Kim Lauenroth
Marc Lohmann
Markus Luckey
Thorsten Merten
Andreas Metzger
Sebastian Middeke
Benjamin Nagel
Pit Pietsch
Claudia Priesterjahn
Rumyana Proynova
Mark-Oliver Reiser
Eugen Reiswich
Holger Rendel
Maik Schmidt
Axel Schmolitzky
Jürgen Schwarz
Ernst Sikorra
David Trachtenherz
Dietrich Travkin
Konrad Voigt
Sven Wenzel

Offizieller Veranstalter

Fachbereich Softwaretechnik der Gesellschaft für Informatik (GI)

Mitveranstalter

s-lab – Software Quality Lab, Paderborn
Universität Paderborn

Unterstützt wird die Tagung zudem von

Schweizer Informatik Gesellschaft (SI)
Österreichische Computer Gesellschaft (OCG)

Sponsoren

SE 2010 Goldsponsoren

SE 2010 Silbersponsoren

SE 2010 Unterstützer

Inhaltsverzeichnis

Eingeladene wissenschaftliche Vorträge

Präsentationen des Industrietags

Keynotes des Industrietags

SOA

Varianten Management

Test

Prozessunterstützung

Forschungsarbeiten

Komponentenmodelle

Moderne Architekturstile

Requirements Engineering

Workshops

Eingeladene wissenschaftliche Vorträge

Software-Werkzeuge - Grundlage für effiziente, komplexe Entwicklungsprozesse

Manfred Nagl

Software Engineering, RWTH Aachen
nagl@se.rwth-aachen.de

Entwicklungsprozesse sind eine besondere Spezies von Geschäftsprozessen. Sie erzeugen höchst komplexe Resultate, die aus vielerlei Modellen, Sichten und sonstigen Beschreibungen bestehen, die in Abhängigkeiten zueinander stehen. Dabei besitzen die Abhängigkeitsbeziehungen unterschiedliche Semantik. Modelle und Abhängigkeiten werden auch auf unterschiedlichen Detailstufen betrachtet.

Diese Entwicklungsprodukte entstehen auch nicht in Prozessen, wie sie in den Lehrbüchern stehen, einerseits durchstrukturiert und vorab geplant oder andererseits völlig spontan und ohne Regeln. Stattdessen gibt es eine Grobstruktur, die im laufenden Prozess verfeinert wird, Rücksprünge aufgrund entdeckter Fehler, Iterationen, um einem gewünschten Ergebnis nahe zu kommen, Verzweigungen, da Alternativen ausprobiert werden, große "Sprünge", wenn Wiederverwendung auf Prozess- oder Produktebene genutzt wird.

Gibt es eine präzise, aber auch pragmatische Methodik, die die Natur der Entwicklungsprozesse und ihrer Ergebnisse berücksichtigt? Wie sehen die Werkzeuge aus, die zu der Methodik, den Prozessen und der Struktur von deren Ergebnis passen? Die gewünschte Werkzeugunterstützung ist meist nicht einmal für die einzelnen Modelle und Sichten vorhanden, erst recht nicht für die Übergänge und Zusammenhänge der Teilprozesse und -resultate sowie für das Wechselspiel zwischen Organisation und technischer Arbeit. Kann die Erfahrung der Entwickler genutzt werden, lassen sich die verschieden Kommunikationsformen der Teammitglieder für die geregelte Zusammenarbeit nutzen? Kann man Wiederverwendung durch Werkzeuge überhaupt maßgeblich unterstützen?

Viele Fragen, die in diversen Forschungsprojekten adressiert wurden, die am Lehrstuhl des Vortragenden durchgeführt wurden oder an denen der Lehrstuhl maßgeblich beteiligt war. Der Vortrag versucht, die Erkenntnisse von etwa 30 Jahren Werkzeugbau für die Bereiche Software-Entwicklung bzw. Entwicklung für Ingenieur-Anwendungsbereiche zu vermitteln.

Trotz vieler Forschungsergebnisse haben diese Projekte die obenstehenden Fragen nicht abschließend gelöst. Was bleibt zu tun und wie erreichen wir das gewünschte Ziel?

Präsentationen des Industrietags

Improving Productivity in the Development of Software-based Systems

Frances Paulisch

CT SE SWI
Siemens AG
Otto-Hahn-Ring 6
80200 München
frances.paulisch@siemens.com

Abstract: This paper describes various techniques for improving productivity in the development of software-based systems. These build on having a strong foundation regarding people and processes. On this solid foundation, it is important to do requirements engineering well so that you develop the right functionality, elicit also the non-functional requirements, and have testable requirements. Structure the system to avoid unnecessary complexity and to take advantage of reuse. During the development use an iterative and lean approach to ensure that the product is realized as efficiently as possible. Finally, work at a higher level of abstraction through model-driven approaches and take advantage of automation provide additional improvements to productivity.

1 Overview

This paper describes various techniques for improving productivity in the development of software-based systems. These build on having a strong foundation regarding people and processes. On this solid foundation, it is important to do requirements engineering well so that you develop the right functionality, elicit also the non-functional requirements, and have testable requirements. Structure the system to avoid unnecessary complexity and to take advantage of reuse. During the development use an iterative and lean approach to ensure that the product is realized as efficiently as possible. Finally, work at a higher level of abstraction through model-driven approaches and take advantage of automation provide additional improvements to productivity.

2 Useful Techniques for Improving Productivity

First of all, it is important to have a solid foundation to build on, in particular in regards to people and processes. The "people factor" is very important as it is the most variable and has a strong influence. You need to have experienced and skilled people on the project and also the ability of the team to work well together is very important. Don't underestimate this aspect. Having an open and trustful communication and ensuring that all have the same understanding of the status of the project throughout the lifecycle is very important especially when dealing with uncertainties in a project. Furthermore, having appropriate and well-defined processes in place helps reduce risks and helps make the various responsibilities clear which is important since so much time and effort can be wasted through poor coordination.

Secondly, have a good understanding of the system you will build. Not only the requirements in terms of functionality, but also the non-functional requirements of the system such as performance, reliability, dependability, scalability, etc. Aim to have testable requirements, also for such non-functional requirements. Be aware of which features are most important in terms of customer value, which have the most influence on the architecture of the system, and which ones are likely to change. Use that information to define the architecture of the system. Making the right choices early and ordering the realization appropriately can help to avoid significant rework during the project. Also aim to structure the system so that complexity and dependencies between components are reduced. If one is able to reuse components or use a product-line engineering approach, this is another very effective way to reduce effort and also often increase the quality of a system.

During the development of the system, an iterative approach is often useful to provide feedback and motivation to the whole product development team. An iterative approach helps alleviate some misunderstandings across phases e.g. between product management and the realization roles. If a problem arises, the sooner one knows about it, the better. This often goes hand-in-hand with test-driven development and continuous integration. Use the techniques of "lean development" to analyze the value stream, identify sources of "waste" and consider, taking all aspects into account, how to adjust appropriately to achieve the best overall benefit. In other words, as is in the slogan of lean development "think big, act small, fail fast, learn rapidly".

There has been much progress made in the past ten to twenty years in the area of modern software engineering for example in the areas of model-driven software engineering and domain-specific languages to enable the development teams to work at a higher level of abstraction. Take advantage of the latest proven techniques and tools to improve the productivity of the development of software-based systems.

Qualität sichtbar machen: Ein Erfolgsrezept in moderner Softwareentwicklung

Melanie Späth, Alexander Hofmann

Capgemini sd&m Research
Capgemini sd&m AG
Carl-Wery Str. 42
D-81739 München
melanie.spaeth@capgemini-sdm.com, alexander.hofmann@capgemini-sdm.com

1. Qualitätsmängel sind Pulverfässer

Im Jahr 2008 wurden weniger als ein Drittel aller IT-Projekte erfolgreich abgeschlossen. Grund für das Scheitern der meisten Projekte sind zu spät entdeckte Qualitätsmängel. Häufig werden erst im System- und Abnahmetest Hunderte von Fehlern gefunden. Die Zeit bis zur Produktivnahme ist dann oft zu knapp, um noch sinnvoll reagieren zu können. Große wirtschaftliche Schäden auf Kundenseite sind oft die Folge.

Dabei schleichen sich Qualitätsmängel über den Entwicklungszyklus hinweg oft unbemerkt ein und bleiben lange unentdeckt: Missverständliche Anforderungen und Lücken in der Spezifikation werden individuell interpretiert, fehlende Performanzvorgaben führen zu falschen Architekturentscheidungen, mangelnde Architekturvorgaben mittelfristig zu Wartungskatastrophen und unfokussierte Entwicklertests zu unvorhergesehen hohem Testaufwand im Systemtest.

Spät gefundene Fehler sind demnach nicht nur teuer: Unentdeckte Qualitätsdefizite in den Anforderungen, der Spezifikation oder der Architektur wirken wie verborgene Pulverfässer und können Projekte sehr plötzlich zum Scheitern bringen.

2. Qualitätssteuerung wird zu einer Kernkompetenz

Kunden fordern deshalb vermehrt von ihren Software-Dienstleistern, dass diese ihre Qualitätssicherung im Griff haben und Ergebnisqualität auch nachweisen können. Sie verlangen zudem einen frühzeitigen Einblick in die Qualität der entstehenden Software.

Die Qualitätssteuerung in Software-Projekten, also die kontinuierliche Messung und Überprüfung der Software-Qualität, bildet sich damit für die Software-Hersteller als neue Kernkompetenz in der Projektabwicklung heraus. Sie fordert in der Praxis anwendbare Methoden und Werkzeuge, um Qualität besser quantifizieren zu können und den Reifegrad transparenter zu machen.

3. Qualität sichtbar machen mit Quasar Analytics®

Wir haben deshalb Quasar Analytics® als Gesamtansatz zur Qualitätssteuerung entwickelt. Hierbei war unser zentrales Ziel, Qualität sichtbar zu machen – und zwar für alle im Projekt Beteiligten. Dies haben wir geschafft durch ein zugrunde liegendes Qualitätsmodell und darauf basierende Methodik- und Toolbausteine.

Das Qualitätsmodell bündelt die für unsere Softwareprojekte relevanten Eigenschaften und Kennzahlen. Es steht damit im Zentrum der strategischen Entscheidungen zur Qualitätssicherung, die zu Beginn eines Projektes getroffen werden. Ausgehend von den Zielen und Rahmenbedingungen eines Projektes werden definierte Qualitätsmerkmale priorisiert. Hierbei werden nicht nur Qualitätsmerkmale von Software, sondern auch die der frühen Ergebnisartefakte mit einbezogen, beispielsweise die Struktur und Verständlichkeit der Spezifikation oder die Rückverfolgbarkeit von Architekturentscheidungen. Abhängig von den gesetzten Prioritäten werden die für das Projekt passenden Methodik- und Toolbausteine für die Qualitätssteuerung ausgewählt.

Die Quasar Analytics® Bausteine ordnen sich in drei Bereiche ein: Review-basierte Prüfungen, Software-Messung und Test.

Review-basierte Prüfungen sichern die Tragfähigkeit und inhaltliche Reife von früh im Projektzyklus entstehenden Artefakten, wie der Systemspezifikation oder -architektur, systematisch anhand von Checklisten und definierten Prüfmethoden ab.

Die Software-Messung konzentriert sich dagegen auf Aspekte der inneren Codequalität, allen voran Wartbarkeit. Definierte Kennzahlen, Erkennungsmuster und Indikatoren spannen den Rahmen für ein konsequentes Monitoring auf. Durch ein „Software-Cockpit“ wird die innere Codequalität nicht nur sichtbar, sondern auch stetig verfolgbar.

Durch strukturiertes Testen sichern wir schließlich die Funktionalität der Software ab, sowie alle testbaren und für das jeweilige Projekt relevanten nichtfunktionalen Merkmale wie Performanz oder Benutzbarkeit. Um dabei Fehler möglichst früh im Prozess zu finden, sind ein klarer Teststufen-Schnitt und zugleich eine risikobasierte und Teststufen-übergreifende Teststrategie wesentliche Voraussetzungen.

4. Fazit

Aufgrund der steigenden Anforderungen an die Softwareentwicklung wird sich Qualitätssteuerung zu einer Kernkompetenz von IT-Dienstleistern entwickeln. Qualitätsrisiken müssen früh erkannt werden, um rechtzeitig gegensteuern zu können. Hierzu liefern wir mit Quasar Analytics Methoden und Werkzeuge, die Qualität im Projekt sichtbar und damit steuerbar machen. Denn nur wer Zeit, Budget und Ergebnisqualität gleichermaßen im Griff hat, kann auch zukünftig nachvollziehbar erfolgreiche Projekte machen.

Stammdatenverwaltung als Basis für eine Service-orientierte IT-Architektur: Ein subjektiver Projektbericht

Armin Bäumker, Jürgen Krüll, Carsten Kunert

syskoplan AG
Bartholomäusweg 26
D-33334 Gütersloh
armin.baeumker@syskoplan.de
carsten.kunert@syskoplan.de

SOA (Service Oriented Architecture) hat sich in den vergangenen Jahren als Paradigma für den Entwurf von IT-Architekturen (zumindest in der Theorie) durchgesetzt. Viele Unternehmen haben dieses in der Praxis aufgegriffen und entsprechende Initiativen mit mehr oder weniger Erfolg gestartet bzw. zu Ende gebracht.

Die Schwierigkeiten, die hierbei auftreten, sind weniger durch technische Probleme sondern eher durch konzeptionelle und organisatorische Mängel charakterisiert. Ein wesentlicher Punkt hierbei ist sicherlich, Bereiche zu identifizieren, für die der Einsatz von SOA angemessen ist und Nutzen verspricht.

Ein Bereich, der sich für einen serviceorientieren Ansatz anbietet, ist die zentrale Verwaltung von Geschäftspartner-Stammdaten. Die erforderlichen Dienste haben eine passende Granularität und erschließen sich einer SOA Architektur in kanonischer Weise. Entscheidend ist aber folgendes: Neben ihrem „selbständigen“ Nutzen sind diese Stammdatendienste Grundlage für den Aufbau weiterer Services, insbesondere im Hinblick auf eine kundenorientiertes Geschäftsmodell des Unternehmens.

In einem aktuellen Projekt (Automobilindustrie: Neuwagengeschäft, Sales- und Aftersales-Prozesse) wurden, neben anderen Systemen, ein zentrales Geschäftspartnersystem eingeführt wurde und über eine neue SOA-Architektur in das neu zu schaffende bzw. vorhandene IT-Umfeld integriert wurde. Das System bietet

- zentrale Datenstrukturen für Geschäftspartnerstammdaten,
- Services zur zentralen Anlage bzw. Änderung von Geschäftspartnerdaten,
- Verteilung von neuen oder geänderten Geschäftspartnerdaten,
- Suchen von Geschäftspartnerdaten,
- Zuordnung einer zentralen ID-Nummer,
- Services zur Sicherstellung der Datenqualität (Adressprüfung, Dublettenprüfung).

Mit diesen Services gelingt es dem zentralen Geschäftspartnersystem einen bereinigten dublettenfreien Satz von Stammdaten zur Verfügung zu stellen. Durch die Verteilung einer einheitlichen Geschäftspartnernummer in die beteiligten Systeme der Landschaft hat man damit eine wichtige Grundlage für das Kundenbeziehungsmanagement (CRM, Customer Relationship Management) und weitere Integrationsmöglichkeiten geschaffen.

- Mehr Möglichkeiten für ein CRM:

  - Die im Kundenlebenszyklus gesammelten Informationen können zusammengebracht und gesamthaft ausgewertet werden und bieten so mehr Möglichkeiten für ein wirksames CRM.
  - Mehr Qualität in der Kundenansprache durch bessere Qualität der Stammdaten (z.B. keine Doppelansprachen aufgrund von Dubletten)

- Mehr Möglichkeiten für Integration und den weiteren Aufbau der SOA:

  - Einfacherer Integration von Applikationen und Informationen (BI). Wegen zentraler ID-Nummer wird Mapping von Daten einfacher.
  - Aufbau höherwertiger Geschäftspartner-Services ist nur auf Basis solch grundlegender Services möglich.

Der Vortrag gibt einen Überblick über den technischen Aufbau der Lösung und beleuchtet auch die kritischen Erfolgsfaktoren, die eher im konzeptionellen und organisatorischen Bereich liegen.

Zur syskoplan AG: Seit 25 Jahren realisiert die syskoplan-Gruppe innovative IT-Lösungen. Dabei werden adaptive und agile IT-Plattformen genutzt und um kundenspezifische Komponenten erweitert. Eine wesentliches Fokusthema sind CRM-Lösungen basierend auf Standardsoftware.

SOA bis in die Präsentationsschicht im Prozessportal einer Leasinggesellschaft

Armin Vogt

Competence Center Technology
S&N AG
Klingenderstraße 5
33100 Paderborn

avogt@s-und-n.de

Abstract: Der Kreditmanager3 – KM3 – basiert auf einer Architektur, die den SOA Gedanken in das Präsentationsschicht erweitert, indem mittels Portlet-Technologie Dialoge samt ihrer Logik und Datenhaltung wieder verwendbar werden.

1 Ausgangssituation

Die S&N AG mit Hauptsitz in Paderborn bedient Kunden vornehmlich aus der Finanzindustrie seit 1991 mit innovativen Individuallösungen und technologischem sowie fachlichem Beratungsleistungen.

Für eine der großen Leasinggesellschaften Deutschlands haben wir deren Kreditentscheidungsprozess analysiert und danach eine vollständig neu entwickelte Software geformt: KM3

KM3 ist eine Web-Applikation, die dem Benutzer eine am Arbeitsablauf orientierte Sicht auf seine Kreditanträge und weitere Vorgänge präsentiert. Herausforderung war einerseits die Ablösung gleich mehrerer „Alt"-Applikationen als auch die Einführung einer neuen Plattform für vorgangsorientierte Sachbearbeitung. Diese Plattform ist nicht fachlich auf den Kreditprozess begrenzt, sondern bündelt sämtliche technischen Aspekte eines sog. Prozessportals. Sie realisiert große Teile des Kundennutzens, indem sie eine Vereinheitlichung jenseits der fachlichen Funktionalität erreicht.

2 Ziele

Die Realisierung des KM3 vollzog sich in einer 2-jährigen Projektlaufzeit, die mit einer intensiven Workshop-Phase begann.

Als Ziel wurde recht früh ausgegeben: Wiederverwendbarkeit von Dialogen in verschiedenen fachlichen Vorgängen; Verlässlichkeit der Daten.

Der Vorschlag zum Portal wurde dann aus diesen Zielen abgeleitet. Es erschien uns notwendig, die traditionelle Trennung in Applikationen in Frage zu stellen. Stattdessen sollten fachliche Vorgänge (die in der Prozessanalyse beschrieben worden waren), als solche im Portal in Erscheinung treten. Der Vorgang kann sich dann in seiner Implementierung verschiedener Dialoge bedienen. Die Verlässlichkeit der Daten meint vor allem zeitliche Aktualität dieser.

3 Lösungsansatz

KM3 übernimmt den SOA Gedanken in seine Architektur, die als erstes den fachlichen Baustein als Ausbringungseinheit (Deployment, Versionierung, Beauftragung) definiert.

Ein Baustein wird als JEE-EAR Datei erzeugt und enthält in sich alle Ebenen einer 3-Tier-Architektur: Präsentationsschicht (eine WAR Datei mit JSR-168-Portlets), Logikschicht (EJB3 Session Beans) und Datenschicht (JPA Mapping; Adapter zu externen Schnittstellen). Als Basistechnologie vertrauen wir auf den JBoss Portal Server, der die Kombination von Dialogen aus verschiedenen Bausteinen in einem Vorgang möglich macht. KM3 wurde folgerichtig nicht als eine Applikation ausgeliefert, sondern als eine Plattform (Portal, Workflow-Steuerung, Deployment), auf der gleichförmige Software-Pakete – sog. Fachliche Bausteine - ausgebracht und miteinander in Aktion tretend ihre fachliche Leistung entfalten.

Während Applikationen sich in ihrer Abgrenzung an Release-Terminen auf Auftragserteilung orientieren, grenzt ein fachlicher Baustein ein konkretes Thema ab und bündelt sämtliche dazu anfallenden Aufgaben.

4 Ergebnisse und nächste Schritte

KM3 ist im November 2009 in Produktion gegangen und löste die bisherige Lotus Notes-Lösung ab. Die agile Vorgehensweise half uns bei der vergleichsweise kurzen Implementierungsphase von 1,5 Jahren, termingerecht zu liefern.

Weitere Prozesse werden als Vorgänge implementiert und sollen in hohem Maße die bereits vorhandenen Dialoge und Services wiederverwenden. Die Plattform wird sich mit ihrem Programmiermodell bewähren müssen, die Abstraktion der Vorgangsorientierung soll eine kostengünstige Weiterentwicklung erreichen.

Software-Varianten im Griff mit textuellen DSLs

Erfahrungsbericht

Johannes Reitzner

CoC Model Driven Development
msg systems ag
Robert-Bürkle-Straße 1
85737 Ismaning
johannes.reitzner@msg-systems.com

Abstract:

Anhand eines Projektes aus der Automobilindustrie wird gezeigt, wie textuelle DSLs zur Beherrschung von länderspezifischen Varianten eines Anwendungssystems eingesetzt werden können. Bei konventionellen Modellierungs- und Programmieransätzen besteht in solchen Fällen die Gefahr vieler Softwareversionen. Die Weiterentwicklung und Wartung wird dabei mit jedem zusätzlichen Land erschwert. Mit aktivem Management der Länderspezifika in Featuremodellen, kombiniert mit geeigneten DSLs und Tools, kann die Entwicklung weiterer Ländervarianten deutlich beschleunigt werden.

1 Ausgangssituation

Für einen Automobilhersteller soll zur Unterstützung des Planungsprozesses von Bestellvorschlägen ein Anwendungssystem erstellt werden, das in mehrere Länder ausgerollt werden soll. Die Planungskernfunktionalität ist für alle Länder gleich, aber es gibt länderspezifische Anforderungen, die sich in geänderten fachlichen Abläufen, Datenstrukturen und Benutzeroberflächen widerspiegeln.

2 Ziele der technischen Umsetzung

Wesentliches Ziel bei der Konzeption und technischen Umsetzung ist es aus einer einzigen Codebasis für jede Landesvariante ein Softwaresystem erstellen zu können, das keine Codeteile enthält, die nur für andere Ländervarianten relevant sind.

Damit wird ein weiteres Ziel unterstützt, nämlich die speziellen Anforderungen und die damit verbundenen technischen Auswirkungen für später neu hinzukommende Länder aktiv managen und kontrollieren zu können.

3 Lösungsansatz und Vorgehen

Zur technischen Unterstützung des aktiven Variantenmanagements wurde ein modell- und generatorgetriebener Lösungsansatz gewählt.

Sämtliche Anforderungen aller Länder werden dahingehend untersucht, ob sich sogenannte Variationspunkte identifizieren lassen, an denen sich ein Landessystem unterschiedlich zu einem anderen verhält. Diese Variationspunkte werden in einem Featuremodell mit dem Eclipse-Werkzeug pure::variants modelliert und gepflegt. Aus dem Featuremodell wiederum entstehen durch Selektion bestimmter Features landesspezifische Variantenmodelle.

Die Brücke zur Technik wird durch eine TMF/Xtext basierte Architektur- und Design spezifische Sprache (ADSL) geschlagen, die komponentenorientierten Entwurf und direkte Bezüge von Architekturartefakten zum Featuremodell ermöglicht.

Codegeneratoren erzeugen aus der Kombination von ADSL- und Featuremodell wesentliche Teile der allgemeinen Codebasis, aus der durch Auswertung eines Variantenmodells das zugehörige landesspezifische System durch automatisierte Tailoring-Workflows „zurechtgeschnitten“ und anschließend assembliert wird.

4 Erfahrungen

Der skizzierte Entwicklungsweg hat sich als praktikabel und effizient erwiesen.

Wesentlich dazu beigetragen hat der Aufbau der benötigten Werkzeugkette als durchgängige, vollständig in Eclipse integrierte Workbench, die Toolbrüche vermeidet. Diese umfasst neben pure::variants, Xtext und openArchitectureWare weitere Eclipse-Plugins für das featureabhängige Zurechtschneiden von manuell erstelltem bzw. generiertem Code.

Die Möglichkeit Modelle schon frühzeitig während ihrer Erstellung IDE gestützt validieren zu können, steigert die Effizienz im Projekt durch weniger fehlschlagende Modellauswertungsläufe.

Verbesserungspotenzial gibt es bei der Unterstützung der Entwickler während der manuellen Erstellung von variantenabhängigen Codeteilen. Hier wären Code-Sichten innerhalb der IDE sehr hilfreich, die dem Zustand nach dem Code-Tailoring auf Basis eines Variantenmodells entsprechen.

Für das Entwicklungsteam hat sich die grafische Visualisierung der textuellen DSL-Modelle als sehr wichtig erwiesen. Diese Funktionalität wird von einem neu entwickelten Eclipse-Plugin bereit gestellt, das durch leicht zu schreibende Modell zu Modell Transformationen für beliebige EMF-Modelle allgemein nutzbar ist.

Verteiltes Testen heterogener Systemlandschaften bei arvato services

Thomas von der Maßen[1], Andreas Wübbeke[2]

[1]IT Management-DCQ
arvato services
An der Autobahn
33310 Gütersloh
Thomas.vonderMassen@bertelsmann.de

[2]Software Quality Lab (s-lab)
Universität Paderborn
Warburgerstraße 100
33098 Paderborn
awuebbeke@s-lab.upb.de

Abstract: arvato services erstellt kundenindividuelle Lösungen durch Entwicklung, Anpassung und Integration verschiedenster Soft- und Hardwaresystem zu heterogenen Systemlandschaften. Die einzelnen Teilsysteme der Lösungen unterscheiden sich dabei in unterschiedlichen Dimensionen, die insbesondere beim Systemintegrationstest des Systemverbundes berücksichtigt werden müssen. Dieser Beitrag beschreibt zum einen die zu berücksichtigenden Dimensionen und weiterhin das von arvato services gewählte Vorgehen um diesen Herausforderungen zu begegnen.

1 Einleitung

arvato services bietet kundenindividuelle Lösungen im Bereich von Informationssystemen für die Bereiche CRM, Fulfillment und E-Commerce an. Der Aufbau eines solchen kundenindividuellen Informationssystems geschieht bei arvato nicht auf Basis eines einzelnen, monolithischen Systems. Vielmehr werden mehrere zuvor selbst entwickelte oder angepasste Teilsysteme integriert, um die kundenindividuellen Anforderungen zu erfüllen [MW09]. Diese Teilsysteme unterscheiden sich in vielerlei Dimensionen. Diese Dimensionen und ihre möglichen Ausprägungen müssen bei der Entwicklung und Integration, aber auch später im Systemintegrationstest berücksichtigt werden. In Kapitel 2 werden die einzelnen Dimensionen mit ihren möglichen Ausprägungen und deren Auswirkungen auf das Testvorgehen beschrieben. Kapitel 3 fasst den Beitrag zusammen.

2 Test heterogener Systemlandschaften

Das Testvorgehen bei arvato services lehnt sich an den fundamentalen Testprozess aus [SL05] an. Im Rahmen der Konzeption des Testvorgehens für Systemlandschaften bei arvato services wurden die in Tabelle 1 genannten Dimensionen als relevant identifiziert. Dabei entstehen für die am Testprozess beteiligten Rollen, wie Testmanager, Testdesigner oder Tester zahlreiche Herausforderungen, die in den jeweiligen Testphasen bzw. Testaktivitäten berücksichtigt werden müssen. Die Auswirkungen der Dimensionen auf die Testphasen und –aktivitäten sind ebenfalls in Tabelle 1 dokumentiert.

Dimension	Begründung / Bemerkung	Auswirkung auf
Organisation	Zu testende Teilsysteme werden von unterschiedlichen Organisationen oder Abteilungen verantwortet	Testplanung, Testkoordination
Standort	Entwicklungsstandorte und Standorte, die bei einem verteilten Systemtest beteiligt sind	Testplanung, Testkoordination
Technologie	Die eingesetzte Entwicklungstechnologie	Testdesign (Testart, Testauswahlkriterien), Werkzeuge
Softwaretyp	Unterscheidung von Standardsoftware, angepasste Standardsoftware und Individualentwicklung	Teststrategie, Testdesign (Testart, Testauswahlkriterien), Werkzeugunterstützung
Systemtyp	Unterscheidung von Informations- und eingebetteten Systemen	Teststrategie, Testdesign (Testart, Testauswahlkriterien), Werkzeugunterstützung
Prüfobjekte	Berücksichtigung unterschiedlicher Prüfobjekte, wie GUI-Ausgaben, Dokumente / Dateien, Datenbankeinträge, …	Testplanung, Teststrategie, Testdesign (Testart, Testauswahlkriterien), Werkzeugunterstützung
Tester-Expertise	Berücksichtigung unterschiedlicher Testexpertise bei Testern (Test-/QS-Experte, Domänenexperte, IT-Experte)	Testdesign, Testkoordination

Tabelle 1: Dimensionen heterogener Systemlandschaften

Unter Berücksichtigung der oben genannten Dimensionen, ihre spezifischen Ausprägungen in einem konkreten Testprojekt sowie deren Auswirkungen müssen frühzeitig in der Planung und Konzeption eines Testprojekts berücksichtigt werden. arvato services greift hierbei auf ein Testframework zurück, welches unterschiedliche Werkzeuge bereitstellt um den Testprozess zu unterstützen. Hierzu zählt beispielsweise ein geeignetes Testmanagement- und Testdesignwerkzeug, um beispielsweise spezifizierte Testfälle örtlich verteilten Reviewern zu Prüfung oder Testern zur Testdurchführung zur Verfügung zu stellen. Des Weiteren werden entsprechende Kommunikationstechnologien, wie Telefon oder Videokonferenzsysteme benötigt.

3 Zusammenfassung

Das Testvorgehen für Informationssysteme auf Basis von heterogenen Systemlandschaften stellt einige Herausforderungen an das Testvorgehen und die einzelnen daran beteiligten Rollen. In diesem Beitrag wurden die dafür relevanten Dimensionen identifiziert und in Bezug auf diese Herausforderungen für einzelne Rollen im Testprozess skizziert.

Literaturverzeichnis

[SL05] Spillner, A.; Linz, T: Basiswissen Softwaretest. dpunkt.verlag, Heidelberg, 2005.

[MW09] von der Maßen, T.; Wübbeke, A.: Lösungsorientierte Software Produktlinienentwicklung in heterogenen Systemlandschaften. In Proc. Produktlinien im Kontext (PIK09), 2009..

Produktlinien-Engineering im SOA-Kontext

Roger Zacharias

Wincor Nixdorf
Corporate Function Chief Technology Officer (WN CF CTO) –
Corporate Architecture Management
Heinz-Nixdorf Ring 1
33106 Paderborn
roger.zacharias@wincor-nixdorf.com

Die Industrialisierung der Softwareproduktion durch Produktlinienkonzepte verspricht Vorteile wie drastische Kostenreduktion, kürzere Time-to-market, Qualitätssteigerung, Effizienzsteigerung und verbesserte Ressourcen-Flexibilität. Durch die Service-Orientierung und Service-orientierte Architekturen (SOA) sollen ähnliche Vorteile ermöglicht werden. Damit stellt sich natürlich die Frage, warum nicht jedes Softwareunternehmen diese Ansätze verfolgt und warum diese nicht bereits weit verbreiteter Standard innerhalb der Softwareentwicklung sind. Dieser Vortrag stellt die technischen und vor allem organisatorischen Herausforderungen und Rahmenbedingungen vor, welche zwingend erfüllt sein müssen, damit eine Softwareproduktlinie im SOA-Kontext die angeführten Versprechen auch einlösen kann.

Hintergründe und Definition

Bei entsprechend hohem Bedarf durch die Konsumenten entwickelt sich jeder neue Industriezweig über verschiedene Stufen vom Künstlertum über das Handwerk zur Industrieproduktion. Jede Industrialisierung zeichnet sich hierbei durch Steigerung der folgenden fünf Kernfaktoren aus: 1. Standardisierung / Kommoditisierung, 2. Wiederverwendung, 3. Arbeitsteilung / Spezialisierung, 4. Automatisierung und 5. Verringerung der Fertigungstiefe. Im Bereich der Softwareentwicklung kann eine Softwareproduktlinie genau diese Faktoren der Industrialisierung maximal unterstützen. Bei einer Softwareproduktlinie handelt es sich hierbei per Definition um eine Gruppe softwareintensiver Systeme, welche eine Menge identischer gemanagter Funktionalitäten bzw. Komponenten teilen, in einer gemeinsamen Zielmarktdomäne angesiedelt sind und auf Basis einer gemeinsamen Menge von Basiskomponenten auf identische Art und Weise entwickelt werden (*vgl. Software Engineering Institute*). Anders gesagt handelt es sich bei Softwareproduktlinien um ein Software-Engineering-Konzept zur effizienten Erstellung eines Portfolios verwandter Softwaresysteme auf Basis einer Menge gemeinsam genutzter Assets, einer gemeinsam genutzten Produktionsinfrastruktur und Produktionsprozessen, verbunden mit einer konsequenten Governance. Der Aufbau einer solchen Produktlinie ist mit diversen technischen und organisatorischen Herausforderungen verbunden:

Technische Herausforderungen

- *Produktlinien-Referenzarchitektur:* Neben den bekannten Architekturprinzipien und -qualitäten muss die Produktlinien-Referenzarchitektur insbesondere eine fachlich unabhängige schmale Produktplattform und eine sehr saubere Komponenten-orientierung in allen Subsystemen und Tiers unterstützen, so dass identische Komponenten in unterschiedlichen Produkten wiederverwendet werden können. Im Bereich der Geschäftslogik eignen sich Fachkomponenten (*siehe GI KobAS - Arbeitskreis 5.10.3*) in Kombination mit SOA-Prinzipien hervorragend.

- *Produktionsinfrastruktur*: Die Produktionsinfrastruktur im Produktlinienumfeld unterscheidet sich gegenüber einer Einzelprodukt-Produktionsinfrastruktur wenig. Die Herausforderung besteht hierbei in der Anpassung der nicht für diesen Einsatz optimierten Werkzeuge für den Produktlinien-Betrieb. Wertvolle Skaleneffekte können zusätzlich durch einen hohen Automatisierungsgrad (z.B. durch DSLs, MDSD) im Produktionsprozess erreicht werden.

Organisatorische Herausforderungen

- *Aufbau- und Ablauforganisation*: Der größte Unterschied zur Einzelprodukt-entwicklung besteht in der Organisation einer Produktlinie. Hier sind vor allem die Trennung in einen Domain Engineering Prozess und einen Application Engineering Prozess, der Aufbau von spezialisierten Teams (Plattformteam, Infrastrukturteam, Architekturteam, Produktteams) und spezifischen Rollen sowie ein komplexer Planungsprozess zu nennen.

- *Governance*: Die zentrale Herausforderung im Produktlinien-Engineering ist eine strikte Governance des Produktionsprozesses und aller Produktlinien-Assets. Hierfür sind geeignete Prozesse (z.B. für Asset-Freigabe, Reuse-Check, Finanzierung), Werkzeuge (wie Komponenten- und Service-Repositories) und natürlich entsprechende Disziplin der Teammitglieder notwendig.

Wincor Nixdorf AG

Wincor Nixdorf ist einer der weltweit führenden Anbieter von IT-Lösungen und -Services für Retailbanken und Handelsunternehmen. Wincor Nixdorf beschäftigt mehr als 9000 Mitarbeiter und ist in rund 100 Ländern präsent, davon in 41 mit eigenen Tochtergesellschaften.

Roger Zacharias war von 2004-2009 Produktlinien-Projektleiter und Produktlinien-Chef-Architekt einer umfangreichen Softwareproduktlinie. Heute vertritt er als Chief Enterprise Architect den Bereich WN Chief Technology Officer (CTO) – Corporate Architecture Management bei Wincor Nixdorf.

Funktionaler Black-Box-Softwaretest für aktive kamerabasierte Fahrerassistenzsysteme im Automotive Umfeld

Florian Schmidt, Nico Hartmann

electronics solutions
MBtech Group
Kolumbusstr. 2
71063 Sindelfingen
florian.schmidt@mbtech-group.com
nico.hartmann@mbtech-group.com

Abstract: In diesem Beitrag wird das funktionale Testen der Embedded Software von kamerabasierten, aktiv agierenden Fahrerassistenzsystemen mit Hilfe einer bilderzeugenden Sensorstimulation für Hardware-in-the-Loop-Tests vorgestellt.

1 Software im Automobil

Moderne Fahrzeuge beinhalten eine Fülle an Embedded Software in derzeit bis zu 80 Steuergeräten oder vernetzt in Steuergeräteverbünden. Die Komplexität ihrer Funktionen steigert sich durch wachsende Kundenerwartungen, erhöhte Anforderungen z. B. an Sicherheitsstandards, und neue technische Möglichkeiten [Gr05, Ch09] kontinuierlich. Aktive Fahrerassistenzsysteme (FAS, engl.: Advanced Driver Assistance Systems, ADAS) unterstützen den Fahrer teilweise durch aktive autonome Eingriffe direkt in die Längs- oder Querregelung des Fahrzeuges (Lenken oder (Not-) Bremsen). Beispiele für umfelderfassende, kamerabasierte ADAS sind Einpark- und Spurhalteassistenten, Fußgänger- und Verkehrszeichenerkennung. Die ständige Interaktion der Software mit den anderen Verkehrsteilnehmern und besonders mit Menschen in ihrer Umgebung legen den benötigten hohen Qualitäts- und Sicherheitsstandard fest. Automotive Software ist also durch hohe Komplexität, komfort- und sicherheitsrelevante fahrzeugspezifische Funktionen und daraus resultierend höchste Anforderungen an die Softwarequalität gekennzeichnet. [Li02]

Daher ist das funktionale Testen der Steuergeräte-Software unerlässlich. Wünschenswert sind eine ausgeprägte Testtiefe und -breite. Es ist dafür erstrebenswert, reproduzierbare und automatisierbare Tests mit der einfachen Möglichkeit der Variation von z. B. Parametern oder Randbedingungen im Dauerbetrieb durchzuführen. Die Hardware-in-the-Loop (HiL) Technologie ermöglicht diese Vorgaben und soll daher als eine Säule bei der Absicherung kamerabasierter Fahrerassistenzsysteme eingesetzt werden.

2 Lösungsansatz

Um kamerabasierte ADAS funktional am HiL zu testen, müssen die Steuergeräte-Eingänge, also die Kamerabilder, simuliert werden. Da es sich üblicherweise um Black-Box-Tests handelt, sind exakt definierbare Auswirkungen der Grafik (oder gar einzelner Sensorpixel) nicht bekannt. Da zusätzlich die Funktionen häufig nur qualitativ beschrieben sind, müssen die simulierten Bilder der Realität möglichst gut entsprechen. Die Herausforderung liegt darin, funktionale Tests unter Echtzeitbedingungen und auf üblichen Testsystemen, aber mit fotorealistischer grafischer Darstellung durchzuführen.

Nach einer Analyse der Anforderungen an das Testen üblicher ADAS-Steuergeräte und an mögliche Graphic Engines folgte die Implementierung. Die hier vorgestellte Lösung ermöglicht zum einen das Erstellen von Testfällen in einem intuitiven Editor, zum anderen die fotorealistische Darstellung der erzeugten Szenarien durch eine 3D Graphic Engine und deren Übertragung auf die (Video-) Schnittstelle des System under Test (SuT). Dabei können Objektbewegungen entweder anhand von Bewegungspfaden vorgegeben oder extern, z. B. durch realistische Fahrzeugmodelle ferngesteuert werden. Die Parametrierung und Steuerung von Objektpositionen und beispielsweise Wettereffekten ist wichtig, um während des sequenziellen Ablaufs von Testfällen die dargestellte Umgebung ändern zu können. Der gleiche Testfall kann bspw. bei leicht geändertem Sonnenstand deutlich andere Ergebnisse liefern. Damit kann im Sinne identifizierender Tests gerade der kritische Bereich von Fahrzeugfunktionen herausgefunden werden. Die Reaktionen des SuT können wiederum rückgeführt und in Echtzeit dargestellt werden, so dass sich die Regelschleife schließt. Der erreichte Grad des Fotorealismus wird durch Klassifikationsverfahren bewertet.

Offene Schnittstellen sorgen bei dem entwickelten Software-Paket u. a. dafür, dass beliebige reale Strecken und 3D-Objekte importiert oder Objektinformationen zum Dienste einer Sensordatenfusion exportiert werden können. Sie sorgen auch für eine einfache Integration in bestehende Testsysteme. Die Durchführung einer riesigen Testmenge wie bei realen Testfahrten unter Laborbedingungen am HiL ist damit ermöglicht.

Als nächster Schritt ist in Übereinstimmung mit üblichen Testprozessen die Automatisierung und Optimierung der Szenario-Erstellung aus Datenbanken vorhandener Testfall-Abschnitte vorgesehen.

Literaturverzeichnis

[Ch09] Charette, R.: This Car Runs on Code. IEEE Spectrum Online, http://www.spectrum.ieee.org, Feb. 2009.

[Gr05] Grimm, K.: Software-Technologie im Automobil. In (Liggesmeyer, P.; Rombach, D., Hrsg.): Software Engineering eingebetteter Systeme. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2005.

[Li02] Liggesmeyer, P.: Software-Qualität. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2002.

ParTeG - Integrating Model-Based Testing and Model Transformations

Dehla Sokenou
GEBIT Solutions
dehla.sokenou@gebit.de

Stephan Weißleder
Fraunhofer-Institut FIRST
stephan.weissleder@first.fraunhofer.de

Abstract: In this paper, we present model-based testing for UML state machines with the test generator ParTeG. We sketch the impact of model transformations on model-based testing and also sketch the results of a corresponding case study with ParTeG.

1 Model-Based Testing

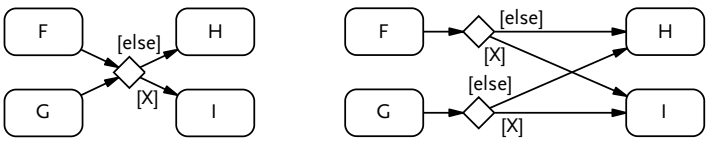
ParTeG [Wei] is a model-based testing tool that was initially developed to implement new algorithms into a prototype as a proof of concept. By now, ParTeG 1.3.1 is available as a free Eclipse plug-in, hosted by Sourceforge. It automatically generates test cases from UML state machines and class diagrams that are annotated with OCL expressions. ParTeG interprets a wide range of OCL expressions, including inequations. ParTeG supports test suite generation for JUnit 3.8 / 4.3, Java Mutation Analysis, and CppUnit 1.12. In the following, we present a short overview of ParTeGs features for model-based test generation.

Coverage criteria are a widely accepted means of test suite quality measurement. There are several kinds of coverage criteria. The most important feature of ParTeG is the ability to satisfy *combined coverage criteria*: For instance, control-flow-based coverage criteria like MC/DC or transition-based coverage criteria like All-Transitions can be combined with boundary-based coverage criteria like Multi-Dimensional: Transition paths are generated according to the control-flow-based respectively transition-based coverage criterion. The selected coverage criterion is converted into a set of model-specific test goals: For instance, All-States is converted into test goals, each of which is referencing one state of the state machine. ParTeG generates paths from the state machine's initial state to the states that are referenced by the test goals. All conditions on each path are transformed and used for concrete input parameter selection corresponding to the selected boundary-based coverage criterion. With these input parameters, the state machine paths are converted into executable test cases of one provided target language. ParTeG also supports test goal monitoring so that already covered test goals are excluded from further test case generations.

Mutation analysis is a wide-spread approach to measure the fault detection capability of a test suite. This approach is based on fault injection in a correct implementation. ParTeG supports mutation analysis by using a mutation factory provided by the tester that delivers a new mutant for each test execution and by generating JUnit code which can be used by the mutation tool Jumble [UTC+07].

2 Model Transformations

Model transformations are means to convert a model into any other model of any other modeling language. ParTeG supports several model transformations to improve the fault detection capability of the generated test suite. It changes the structure of a state machine while preserving its semantics. Several transformations can be found in a corresponding report on an industrial cooperation [Wei09]. We present just one example: State machines can contain choice pseudostates that reference more than one incoming and more than one outgoing transition (see Figure 1(a)). The outgoing transitions also contain guards. Now we compare the effects of existing coverage criteria: Transition-based coverage criteria are focussed on transition sequences. They can enforce that all transition sequences from the states *F* or *G* to *H* or *I* are traversed. They fail, however, if the guard condition is more complicated and we are interested in condition values. Control-flow-based coverage criteria are focussed on guard conditions, but they do not necessarily cover all transition paths. As a consequence, the guard values *[X]* and *[else]* may be tested, e.g. just including the state *G* – all faults that are related to state *F* may be undiscovered. An intuitive solution seems to consist of generating two test suites for both kinds of coverage criteria. As a result, transition sequences and guards are tested. However, the guards are not properly tested for each sequence. Instead, we propose to transform the state machine, e.g. by splitting choice pseudostates according to their number of incoming transitions. Figure 1(b) shows the transformed test model for this example. Since there is only one incoming transition for each choice pseudostate, any selected control-flow-based coverage criterion now has to be satisfied for both states *F* and *G*. In our industrial cooperation, all proposed model transformations resulted in an significant increase of the generated test suite's fault detection capability. All proposed transformations are implemented in ParTeG. They can be automatically executed for all used state machines.



(a) Part of the initial state machine. (b) Part of the transformed state machine.

Figure 1: Model transformation to split choice pseudostates.

References

[UTC+07] Mark Utting, Len Trigg, John G. Cleary, Archmage Irvine, and Tin Pavlinic. Jumble. http://jumble.sourceforge.net/, 2007.

[Wei] Stephan Weißleder. ParTeG (Partition Test Generator). http://parteg.sourceforge.net.

[Wei09] Stephan Weißleder. Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In *Models 2009*, October 2009.

Integrierte Software-Qualitätssicherung des CMS FirstSpirit auf Basis von FirstSpirit

Jörn Bodemann[1], Matthias Book[2]

[1]e-Spirit AG
Barcelonaweg 14
44269 Dortmund
bodemann@e-spirit.de

[2]adesso AG
Stockholmer Allee 24
44269 Dortmund
book@adesso.de

Abstract: Die Entwicklung und Evolution komplexer Softwareprodukte für eine Vielzahl von Plattformen stellt hohe Ansprüche an die effektive und effiziente Qualitätssicherung eines jeden Produkthauses. Als Beispiel für einen Qualitätssicherungsansatz, der sich unter diesen Herausforderungen in der Praxis bewährt hat, beschreibt unser Beitrag, wie das Content Management System FirstSpirit als Basis eines vollintegrierten Qualitätsmanagements seiner selbst eingesetzt wird, sowohl im Rahmen der manuellen und automatischen Testdurchführung wie auch im Release Management.

1 Motivation

FirstSpirit ist ein Enterprise Content Management System (CMS) für High-End-Anwendungen, das neben der redaktionellen Erfassung und Aufbereitung von Inhalten für beliebige Präsentationskanäle umfassende Schnittstellen zur Portal-, Datenbank- und Prozessintegration bietet. Die Komplexität des Systems liegt dabei zum einen in seiner Zwei-Client-Struktur begründet (ein Java- und ein web-basiertes Front-End, deren Masken beide vom Kunden frei konfigurierbar sind), zum anderen in dem umfangreichen Testraum, der durch die Vielzahl möglicher Java Development Kits (JDKs), Betriebssysteme und Datenbanken aufgespannt wird.

2 Einsatz von FirstSpirit zur eigenen Qualitätssicherung

Zur Qualitätssicherung des CMS FirstSpirit setzt das Entwicklungsteam ein breites Spektrum von Testverfahren ein, deren Integration in der Praxis von entscheidender Bedeutung ist [Li09]. Die Besonderheit im Hause e-Spirit ist an dieser Stelle, dass ein Großteil der Tests durch das Produkt selbst unterstützt bzw. sogar automatisiert wird.

In vielen Bereichen, die automatischen Tests nicht mit vertretbarem Aufwand zugänglich sind (z.B. Installationsroutinen, GUI-Darstellung, Lokalisation), bleiben manuelle Tests weiterhin unumgänglich. Die FirstSpirit-Infrastruktur wird hier zur

strukturierten Verwaltung der Testfälle eingesetzt, u.a. zur Formulierung von Vorbedingungen, Handlungen, Erwartungen; zur Organisation von Testfällen in Testsuiten für verschiedene Releases; sowie zur Erfassung von Ergebnissen und der Sammlung statistischer Daten über die Testdurchführung. Dieses Vorgehen trägt nicht nur zur Effizienz der manuellen Testprozesse bei, sondern testet das Produkt FirstSpirit implizit noch einmal auf Systemebene mit.

Im Bereich des Black-Box-Testing unterstützt FirstSpirit darüber hinaus die weitreichende Automatisierung eigener Regressionstests im Kleinen wie im Großen: Um Kunden größtmögliche Flexibilität bei der Verarbeitung von Inhalten zu geben, verfügt das CMS über eine ausgesprochen mächtige Sprache zur Definition von Eingabemasken und Seitentemplates sowie zur Auswertung von Ausdrücken. Im Rahmen von Regressionstests setzen wir die gleiche Sprache ein, um Tests der Template-Syntax, Eingabekomponenten, API-Funktionen und Systemvariablen zu formulieren und ihr Ergebnis zu prüfen. Eine so definierte Sammlung von Testfällen kann automatisch abgearbeitet werden, um z.B. Regressionstests gegen verschiedene Kombinationen von JDKs, Betriebssystemen und Datenbanken zu fahren.

Während die Regressionstests im Kleinen auf bestimmte Komponenten zielen, wird im Rahmen der Regressionstests im Großen das Gesamtsystem automatisiert getestet – genauer die Ausgabegenerierung, die die Kernfunktionalität des CMS darstellt und auf einer Vielzahl von Komponenten aufbaut. Zu diesem Zweck wird die Ausgabe für ein Referenzprojekt mittels FirstSpirit generiert und automatisiert mit der Referenz verglichen, um Unterschiede zwischen Releases und Plattformen aufzudecken.

Auch Continuous Integration und Release Management erfolgen über eine mit FirstSpirit selbst generierte Web-Anwendung, in der z.B. technische Fehlerbeschreibungen nach dem Bugfixing für die Release Notes in die Kundensprache übersetzt werden.

3 Fazit

Das CMS FirstSpirit eignet sich aufgrund der Mächtigkeit seiner integrierten Entwicklungs- und Ausführungsumgebung als Basis für ein vollintegriertes Qualitätsmanagement, das Bug-Reporting, Testdefinition und –auswertung sowie Versionskontrolle und Release-Management für das Produkt mit dem Produkt selbst realisiert. Diese Integration ist von zentraler Bedeutung dafür, der Qualitätssicherung ein exaktes und aktuelles Bild der Issues und Features in einzelnen Versionen zu geben, trägt zur Effizienzsteigerung der Testprozesse bei, und unterzieht das Produkt durch die tägliche intensive Nutzung implizit einem weiteren kontinuierlichen Systemtest, um gleichbleibend hohe Qualität der Releases zu gewährleisten.

Literaturverzeichnis

[Li09] Liggesmeyer, P.: Software-Qualität: Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, Heidelberg/Berlin 2009

Prozessmanagement in der Software-Entwicklung

Philipp Wibbing, André Krick

UNITY AG
Lindberghring 1
33142 Büren
philipp.wibbing@unity.de
andre.krick@unity.de

Abstract: Die effiziente Entwicklung von Software hängt wesentlich von den umgebenden Prozessen ab. Wesentlich sind die praktische Umsetzung von Anforderungsmanagement, Spezifikationsprozess, Implementierung, Test- und Releasemanagement. Eine optimale und praktisch umgesetzte Prozessgestaltung schafft eine gemeinsame Basis der Zusammenarbeit, reduziert unnötige Kommunikationsaufwände und minimiert spät entdeckte Fehler.

1 Einführung

Software-Entwicklungs-Organisationen[1] sehen sich aktuell zunehmenden Herausforderungen gegenüber: Schneller technologischer Wandel, neue Methoden und Konzepte, höherer Integrationsgrad in die bestehende Systemumgebung (insbesondere im Bereich Embedded Software), Intensivierung der Arbeitsteilung (z. T. mit räumlicher Trennung) etc. Ein bestimmender Faktor für die Effizienz, aber auch für die Effektivität ist die in der Praxis umgesetzte Prozessgestaltung.

2 Prozessmanagement in der Software-Entwicklung

Der dargestellten Komplexität kann in effizienter Form nur über anforderungsgerecht gestaltete und in der Praxis angewandte Prozesse begegnet werden. Das Prozessmanagement muss dazu an die konkreten Parameter der Software-Entwicklung angepasst werden. Etwa die eingesetzte Technologie und die Methodik, der Status im Produktlebenszyklus oder die Flexibilitätsanforderungen der Kundenseite bilden Anforderungen an die Prozessgestaltung.

[1] Unter dem Begriff werden sowohl für die Software-Entwicklung zuständige Abteilungen als auch ganze Unternehmen, deren Geschäftszweck die Entwicklung von Software ist, verstanden.

2.1 Anforderungsmanagement und IT Spezifikation

Die Identifikation und stringente Umsetzung von Kunden-Anforderungen ist entscheidend für die effiziente und effektive Erstellung eines Software-Systems. Durch die Prozessgestaltung müssen Anforderungen eindeutig, verständlich[2], vereinzelt und widerspruchsfrei sowie lösungsneutral identifiziert werden. Diese Anforderungen müssen für agile und nicht-agile Vorgehensmodelle formal unterschiedlich in der Prozessgestaltung umgesetzt werden.

Über die Umsetzung der so identifizierten Anforderungen müssen geeignet besetzte Gremien anhand von Wirtschaftlichkeitsbetrachtungen[3] entscheiden. Im Prozess liegt die Herausforderung darin, frühzeitig belastbare Schätzungen der Umsetzungsaufwände zu erhalten; insbesondere die Spezifikation der konkreten Umsetzung allerdings möglichst spät durchzuführen, da diese schon wesentliche Aufwände beinhaltet.

Neben den funktionalen Anforderungen des Kunden muss die Softwaregestaltung weitere Anforderungen umsetzen. Regulatorische Anforderungen umfassen relevante gesetzliche Anforderungen und zu erfüllende Normen. Die IT Strategie (inklusive der Architekturstrategie) beinhalten weitere Anforderungen an die Software.

2.2 Release und Deployment Management

Die transparente Steuerung des Entwicklungs- und Veröffentlichungsprozesses obliegt dem Release und Deployment Management. Die Prozessgestaltung muss Methoden zur Bildung wohldefinierter Releases und deren Planung umfassen. Releases erfüllen eine Teilmenge der umzusetzenden Anforderungen, müssen jedoch jederzeit die Anforderungen allgemeiner Softwarequalitätsmerkmale[4] erfüllen.

2.3 Implementierung

Der Prozess der Implementierung regelt die Zusammenarbeit der Entwickler und setzt Standards für die Codierung um. Integriert werden müssen folglich Programmierrichtlinien, technologische Rahmenbedingungen[5] sowie relevante Paradigmen (z.B. modellgetriebene oder aspektorientierte Entwicklung).

2.4 Testmanagement

Die Gestaltung des Testmanagements ist ein Erfolgsfaktor für die Qualität des Software-Produkts. Die anforderungsgerechte Gestaltung von Testarten, die Aufbauorganisation des Testmanagements sowie die stringente Teststeuerung sind entscheidend.

---

[2] Die Verständlichkeit bezieht sich auf alle am Entwicklungsprozess beteiligten Gruppen. Insbesondere sind hier die Anwendervertreter auf Kundenseite und die Entwickler auf Seiten des Umsetzers gemeint.
[3] Auch als „IT Business Cases“ bezeichnet
[4] Softwarequalitätsmerkmale werden beispielsweise in DIN ISO 9126 definiert
[5] Prozessanforderungen sind beispielsweise für JEE- andere als für Legacy-Plattformen.

How we do it - Business Application Entwicklung mit Oracle ADF

Ulrich Gerkmann-Bartels

TEAM
Partner für Technologie und angewandte Methoden der Informationsverarbeitung GmbH
Hermann-Löns-Str. 88
33104 Paderborn
ugb@team-pb.de

Abstract: Eine grundlegende Herausforderung in unserer individuellen Projektarbeit besteht darin, Use Cases für Fachabteilungen des Kunden zu realisieren, die von vorhandenen Lösungen unzureichend oder nur mit erheblichen Aufwand umzusetzen sind.

1 Rapid (Business) Application Development

Viele individuelle IT- Projekte bestehen heute aus der Aufgabe spezielle Use Cases für eine Fachabteilung zu realisieren, die nicht durch eine allumfassenden Lösung wie SAP oder Oracle eBusiness Suite aus verschiedenen Gründen abdecket werden kann. Häufig ist der Antrieb, dies durch eine kleinere individuelle Umsetzung zu realisieren. Aus unserer Sicht begründet u.a. durch folgende Faktoren: (a) Eigene IT ist überlastete, (b) Integration in die Lösung ist zu langsam oder zu kostenintensiv und (c) Anbindung neuer Schnittstellen, die bisher in den vorhanden Applikationen noch nicht berücksichtigt worden sind.

Aus unserer Sicht ist es zu diesem Zwecke notwendig, eine entsprechende Architektur und Technologie zu Verfügung zu haben, die es uns erlaubt, mit einer Fachabteilung und einem Business Developer eine spezielle Lösung schnell zu realisieren. Wesentlich sind hier die Punkte Architektur und Business Developer.

Die gewählte Architektur muss es erlauben verschiedene Quellen wie bestehende Datenbankschema, CSV – Dateien oder Web Services auf einfache Art und Weise zu verwenden und in einer Schicht zu bündeln, damit diese aus Sicht der Benutzeroberfläche in der Art und Weise einer 4GL Umgebung durch einen Business Developer verwendet werden kann.

Wichtiger Aspekt aus unserer Sicht ist die Verwendung der Standartplattform Java und die komponentenbasierende Benutzeroberflächenentwicklung nach Java EE / Java Server Faces. Aus unser Sicht sollte sich ein Business Developer nicht mit Java Script, AJAX , CSS oder HTML auseinander setzen müssen, um eine Lösung im Sinne des Kunden zu implementieren. Dies überlassen wir den Entwicklungsteams von Frameworks.

2 Metadata Services

Ein wichtiger Punkt innerhalb unserer Technologie – Auswahl für die Realisierung von Business Applikationen, ist der Aspekt, dass das Ergebnis nach unseren Erfahrungen häufig eine Abwandlung oder eine Ausprägung in die eine oder andere Richtung erfährt, so dass man nicht mehr von einer Implementierung eines speziellen Use Case sprechen kann. Hier erheben wir den Anspruch, dass nicht jede Abwandlung oder Ausprägung in eine neue Realisierung eines noch spezielleren Use Case münden sollte, sondern verstehen das als Customization der vorhandenen Realisierung.

Analysiert man auf welcher Ebene eine solche Anwendung anpassbar sein sollte, finden sich schnell folgende Punkte [OR01]: Business Objekte, Kontrollfluss in einer Anwendung (Dialogfolge), Elemente einer Oberfläche bis zu der Anpassung durch den Anwenders.

Für die funktionale Bereitstellung einer solchen Anpassungsmöglichkeit ergeben sich zwei unterschiedliche Bereiche (a) zur Laufzeit durch den Anwender oder Administrator[1] und (b) den Entwickler, der eine bestehende Lösung auf der Ebene der Business Objekte und Kontrollflüsse anpasst.

Die Entscheidung, dass eine solche nicht funktionale Ausprägung innerhalb der Entwicklung einer Business Applikation verfügbar ist, hat Auswirkungen auf die Produktivität innerhalb der eigenen Realisierung. Wir denken, dass dies notwendig ist und vom Markt verlangt wird. Der zweite wichtige Aspekt aus technologischer Sicht ist, dass die von uns gewählte Technologie die Realisierung und Bereitstellung des Customizing durch XML Konfigurationen durchführt. Diese sind vom Menschen lesbar und unabhängig maschinell bearbeitbar.

Literaturverzeichnis

[OR01] Maier, B.; Nimphius, F.: Introduction to Oracle Metadata Services (MDS), DOAG 2009 Präsentation DOAG 2009; Folie 19.

[1] U.a. ist hier auch das aktivieren von Features einer Applikation zu verstehen, die durch den Kunden erstmalig lizenziert wurden.

Forschungsarbeiten

Extending Web Applications with Client and Server Plug-ins[1]

Markus Jahn, Reinhard Wolfinger, Hanspeter Mössenböck

Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz
Altenbergerstr. 69, 4040 Linz, Austria
{jahn, wolfinger, moessenboeck}@ase.jku.at

Abstract: Plug-in frameworks support the development of component-based software that is extensible and customizable to the needs of specific users. However, most current frameworks are targeting single-user rich client applications but do not support plug-in-based web applications which can be extended by end users. We show how an existing plug-in framework (Plux.NET) can be enabled to support multi-user plug-in-based web applications which are dynamically extensible by end users through server-side and client-side extensions.

1 Introduction

Although modern software systems tend to become more and more powerful and feature-rich they are still often felt to be incomplete. It will hardly ever be possible to hit all user requirements out of the box, regardless of how big and complex an application is. One solution to this problem are plug-in frameworks that allow developers to build a thin layer of basic functionality that can be extended by plug-in components and thus tailored to the specific needs of individual users. Despite the success of plug-in frameworks so far, current implementations still suffer from several deficiencies:

(a) *Weak automation*. Host components have to integrate extensions programmatically instead of relying on automatic composition. Furthermore, plug-in frameworks usually have no control over whether, how or when a host looks for extensions.

(b) *Poor dynamic reconfigurability*. Host components integrate extensions only at startup time whereas dynamic addition and removal of components is either not supported or requires special actions in the host.

(c) *Separate configuration files*. Composition is controlled by configuration files which are separated from the code of the plug-ins. This causes extra overhead and may lead to inconsistency problems.

(d) *Limited Web support*. Plug-in frameworks primarily target rich clients or application servers. Although some frameworks extend the plug-in idea to web clients, they are still limited in customization and extensibility: They neither support individual plug-in configurations per user, nor do they integrate plug-ins executed on the client.

[1] This work was supported by the Christian Doppler Research Association and by BMD Systemhaus GmbH.

Over the past few years we developed a client-based plug-in framework called *Plux.NET* which tries to solve the problems described above. While issues (a) to (c) are covered in previous papers [WDP06, WRD08, Wo08, RWG09], this paper deals with issue (d) and describes how Plux.NET can be enabled for multi-user web applications. We show how such web applications can be extended by user-specific plug-ins both on the server side and on the client side. Client-side plug-ins can either run in managed .NET mode [MS08] or in sandbox mode using the Silverlight technology [MS09]. To demonstrate our approach we present a case study, namely a web-based time recorder composed of server-side, client-side, and sandbox components.

Our research was done in cooperation with BMD Systemhaus GmbH, a company offering line-of-business software in the ERP domain. ERP applications consist of many different features that can either be used together or in isolation, thus being an ideal test bed for a plug-in approach.

This paper is organized as follows: Section 2 describes the plug-in framework Plux.NET as the basis of our work. Section 3 uses a case study to explain the architecture of a component-based web application built with Plux.NET. Section 4 compares our work to related research. The paper closes with a summary and a prospect of future work.

2 The Plux.NET framework

Plux.NET [WDPM06, Wo10] is a .NET-based plug-in framework that allows composing applications from plug-in components. It consists of a thin core (140 KBytes) that has slots into which extensions can be plugged. Plugging does not require any programming. The user just drops a plug-in (i.e., a DLL file) into a specific directory, where it will be automatically discovered and plugged into one or several matching slots. Removing a plug-in from the directory will automatically unplug it from the application. Thus adding and removing plug-ins is completely dynamic allowing applications to be reconfigured for different usage scenarios on the fly without restarting the application.

Plug-in components (so-called *extensions*) can have slots and plugs. A *slot* is basically an interface describing some expected functionality and a *plug* belongs to a class implementing this interface and thus providing the functionality. Slots, plugs and extensions are specified declaratively using .NET attributes. Thus the information that is necessary for composition is stored directly in the metadata of interfaces and classes and not in separate XML files as for example in Eclipse [Ec03].

Let's look at an example. Assume that some host component wants to print log messages with time stamps. The logging should be implemented as a separate component that plugs into the host. We first have to define the slot into which the logger can be plugged.

```
[SlotDefinition("Logger")]
[Param("TimeFormat", typeof(string))]
public interface ILogger {
  void Print(string msg);
}
```

The slot definition is a C# interface tagged with a `[SlotDefinition]` attribute specifying the name of the slot ("Logger"). Slots can have parameters defined by `[Param]` at-

tributes. In our case we have one parameter `TimeFormat` of type `string`, which is to be filled by the extension and used by the host. Now, we are going to write an extension that fits into the `Logger` slot:

```
[Extension("ConsoleLogger")]
[Plug("Logger")]
[ParamValue("TimeFormat", "hh:mm:ss")]
public class ConsoleLogger: ILogger {
  public void Print(string msg) {
    Console.WriteLine(msg);
  }
}
```

An extension is a class tagged with an `[Extension]` attribute. It has to implement the interface of the corresponding slot (here `ILogger`). The `[Plug]` attribute defines a plug for this extension that fits into the `Logger` slot. The `[ParamValue]` attribute assigns the value `"hh:mm:ss"` to the parameter `TimeFormat`.

Finally, we implement the host, which is another extension that plugs into the `Startup` slot of the Plux.NET core. The extension has a slot `Logger`; this is specified with a `[Slot]` attribute. This attribute also specifies a method `AddLogger` that will be called when an extension for this slot is plugged. `AddLogger` integrates the logger extension and retrieves the `TimeFormat` parameter.

```
[Extension("MyApp")]
[Plug("Startup")]
[Slot("Logger", OnPlugged="AddLogger")]
public class MyApp: IStartup {
  ILogger logger = null; // the logger extension
  string  timeFormat;    // parameter of the logger extension

  public void Run() {
    ...
    if (logger != null) {
      logger.Print(DateTime.Now.ToString(timeFormat) + ": " + msg);
    }
  }

  public void AddLogger(object s, PlugEventArgs args) {
    logger = (ILogger) args.Extension;
    timeFormat = (string) args.GetParamValue("TimeFormat");
  }
}
```

This is all we have to do. If we compile the interface `ILogger` as well as the classes `ConsoleLogger` and `MyApp` into DLL files and drop them into the plug-in directory everything will fall into place. The Plux.NET runtime will discover the extension `MyApp` and plug it into the `Startup` slot of the core. It will also discover the extension `ConsoleLogger` and plug it into the `Logger` slot of `MyApp`. (see Figure 1)

Figure 1: Composition architecture of the logger example

Plux.NET offers a light-weight way of building plug-in systems. Plug-ins are just classes tagged with metadata. They are self-contained, i.e., they include all the metadata neces-

sary for discovering them and plugging them together automatically. There is no need for separate XML configuration files. The example also shows that Plux.NET is event-based. Plugging, unplugging and other actions of the runtime core raise events to which the programmer can react. The implementation of Plux.NET follows the plug-in approach itself. For example, the discovery mechanism that monitors the plug-in directory is itself a plug-in and can therefore be replaced by some other way of discovery.

Other features of Plux.NET that cannot be discussed here for the lack of space are *lazy loading* of extensions (in order to keep application startup times small), management of *composition rights* (e.g., which extensions are allowed to open a certain slot, and which extensions are allowed to fill it), as well as a *scripting API* that allows experienced developers to override some of the automatic actions of the runtime core. These features are described in more detail in [Wo10].

3 Extending Plux.NET for the Web

The Internet has become fast and ubiquitous enough for implementing software as web applications that can be accessed by multiple clients. Web applications make it easier to bring software to the market. Additionally, updates can be done centrally without bothering administrators in different companies. However, web applications face similar problems as rich-client applications: when they get too big and feature-rich, they become hard to understand and difficult to use. Furthermore, current web applications are hardly customizable and usually not extensible by users. Finally, it is generally not possible to connect the clients' local hardware to web applications. The goal of our research is to find solutions for these problems.

Our idea is to apply the plug-in approach also to web applications by extending Plux.NET so that it becomes web-enabled. Originally, Plux.NET was designed for single-user rich-client applications. In its extended form it supports multi-user web applications where specific users have their individual set of components and their individual composition state. Users will be able to extend web applications with their own plug-ins or with plug-ins from third party developers.

Plux.NET web components can be classified according to their *composition type* and their *visibility scope*. Depending on their visibility, they can affect just a single user, a group of users (e.g., a department of a company), or all users of a web application. Currently, the composition type of a component can be server-side, client-side, and sandbox integration.

Server-side components are installed and executed on the server. Their advantage is that they are smoothly integrated into the server-based web application. They have minimal communication overhead and maximum availability. However, server-side components increase the work load on the server and constitute a security risk. Users need to be authorized to install their extensions on the server.

Client-side components are placed on the client and plugged into a web application virtually. Their major advantage is that users can integrate their local resources (e.g. hardware) into web applications. Another benefit is that users without authorization for in-

stalling plug-ins on the server can still extend a web application. The disadvantages of client-side extensions are the benefits of server-side extensions: Communication between server-side and client-side components causes a certain overhead. Also, client-side extensions are only available when the client is connected.

A third way of composition is to install components on the server, but to execute them in a client-side sandbox such as Adobe Flash or Silverlight (in the case of .NET). This composition type lends itself for building rich user interfaces in a web browser that go beyond the features of HTML or JavaScript, especially if these interfaces should be extensible and customizable without requiring extensions to be installed on the client. Sandbox extensions are copied to the client on demand and are executed there, so they help to keep the work load on the server small.

Even though components are executed on different computers and in different environments, the development and composition process in Plux.NET is the same for server-side plug-ins, client-side plug-ins, and sandbox plug-ins. The composition model is based on the metaphor of slots and plugs as in the rich client approach. Developers just specify the components' metadata in a declarative way. The runtime core is responsible for plug-in discovery, composition, and communication. Therefore, if there are no dependencies on hardware-specific resources it is possible to reuse a rich client component also as a server-side or a client-side component for web applications. For reusing rich client components as Silverlight sandbox components, they need to be recompiled in a special way, because Silverlight assemblies are not binary compatible with other .NET assemblies.

3.1 Case study and scenarios

To demonstrate the idea of our web-enabled plug-in framework this section shows some scenarios: In a case study we extend a web application by various user-specific plug-ins which are composed by using the different composition types described above and by applying the different visibility scopes to them.

Figure 2 shows the component architecture of a web application which is used for recording and evaluating the labour times of employees. The recording and the statistics are implemented by components, each consisting of a GUI component for rendering the user interface and a component for the business logic. The business logic components (*Recorder*, *Statistics*) are connected to the *Data Provider* component while the GUI components are used by the *Layout Manager* component for building the user interface. The *Layout Manager* is plugged into the root component named *Time Recorder*. All components in Figure 2 are server-side components and thus are installed and executed on the server. To keep the scenario simple, some other required components (e.g., Plux.NET runtime components) are hidden in the following pictures.

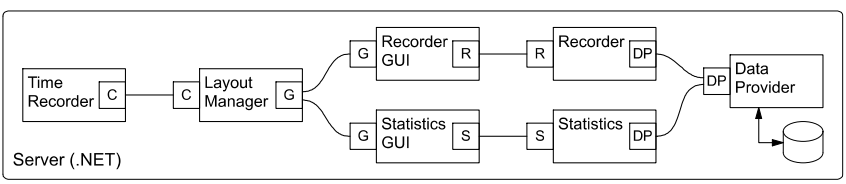


Figure 2: Component architecture of a time recorder web application

To get an impression of how such a web application could look like, Figure 3 shows a possible user interface. The *Layout Manager* arranges the user interfaces of the *Recorder GUI* and *Statistics GUI* components in its window area, depending on some metadata which describes the arrangement of the GUI components. The *Recorder GUI* has buttons for starting, pausing, and stopping time recording while the *Statistics GUI* displays the recorded labour time.

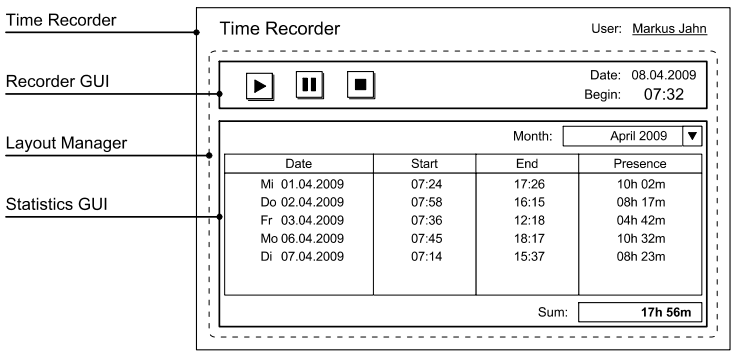


Figure 3: Possible user interface of the time recorder.

Figure 2 contains only server-side components. Thus, the user interface in a client's web browser is restricted to web technologies such as HTML, CSS and JavaScript. However, in our scenario developers want to use the rich internet technology Silverlight for building a more sophisticated user interface. Therefore, they implement the components *Layout Manager*, *Recorder GUI* and *Statistics GUI* as Silverlight components. Since these components are discovered as Silverlight components, they are automatically sent to the client and executed in its Silverlight environment while business logic components stay on the server. Our composition model composes sandbox components in the same way as server-side components. Sandbox components are virtually plugged into server-side components and vice versa. The new composition state is outlined in Figure 4.

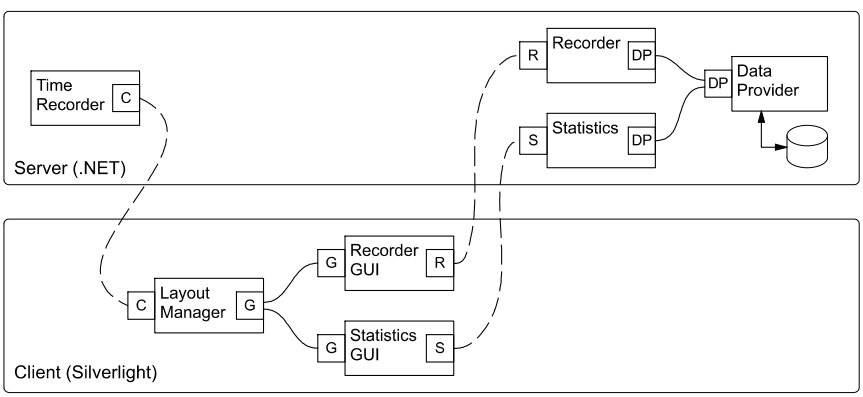


Figure 4: Silverlight components are virtually plugged into server-side components and vice versa

Next, we assume that a user (Client 1) is not fully satisfied with the provided functionality of our time recorder. For annotating his activities during the day he wants to add a

note to each time stamp. Therefore, Client 1 implements his own components for this feature. Since he wants to access his components from any computer, he installs the server-side component *Notes* (a note editor) and the Silverlight component *Notes GUI* on the server. Similar to the other components of our application the server-side component *Note* is used for business logic, while the Silverlight component *Notes GUI* displays the user interface. As Client 1 has set the visibility of these components to private, he is the only user who can see and access them. The individual view of the composition state for Client 1 is shown in Figure 5.

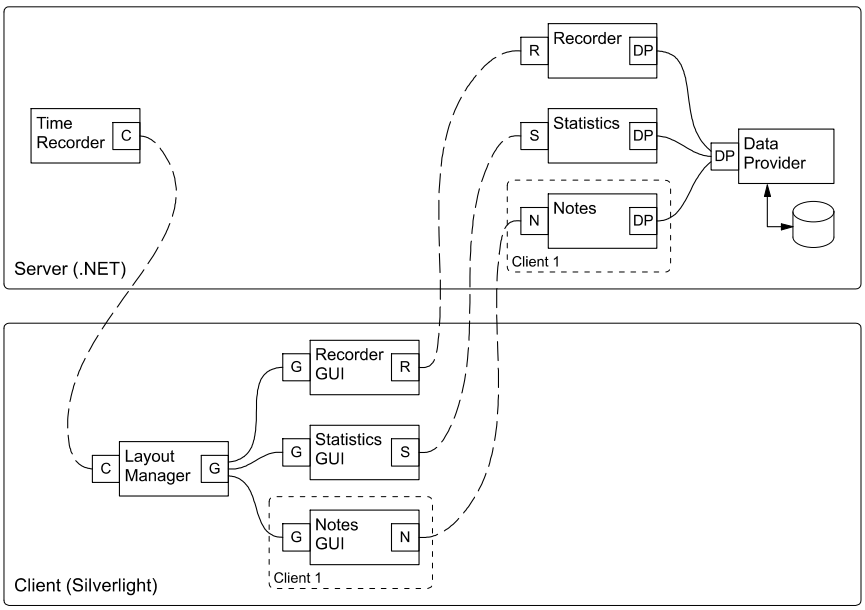


Figure 5: User-specific server-side and sandbox components for Client 1

Besides adding components for a specific user, it is also possible to remove them. For example, a company could deny access to the Silverlight component *Recorder GUI* for several employees. Instead, a hardware time recorder which is represented by a client-side component *Hardware Recorder* could be installed at the company's entrance. The *Hardware Recorder* uses the same server-side *Recorder* component as the Silverlight component *Recorder GUI* did. Figure 6 shows how the Silverlight component *Recorder GUI* is replaced by the client-side component *Hardware Recorder* for clients 2 to 5. Client-side components can be virtually plugged both into server-side components and into Silverlight components and vice versa. If client-side components are connected to Silverlight components the communication between them does not affect the server.

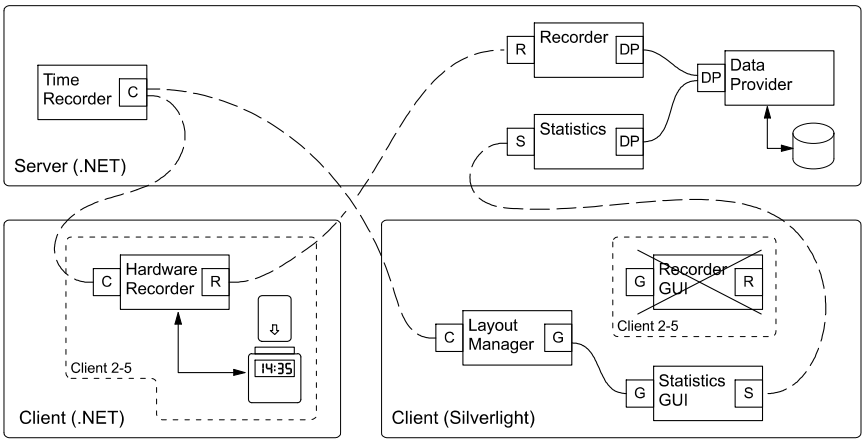


Figure 6: From the view of clients 2 to 5, the Silverlight component *Recorder GUI* is replaced by the client-side component *Hardware Recorder*

3.2 Architecture

We will now look at the architecture and the internals of Plux.NET for web applications. Some aspects described in this section are still under development but the overall architectural design is finished and has been validated with prototypes.

The root component of Plux.NET is the *Runtime* core, which has two slots, one for a discovery component and one for the application's root component. The default discovery component monitors the plug-in directory. Whenever an extension is dropped into this directory its metadata are read and the *Runtime* checks all loaded components for open slots into which the new extension can be plugged. After an extension has been plugged, its own slots are opened and the *Runtime* looks for other extensions (loaded or unloaded) that can fill these slots. These steps are repeated until all matching slots and plugs have been connected.

To perform this composition procedure for multi-user web applications the architecture of Plux.NET had to be extended. We now have different environments in which components can live: There is one environment for server-side components, one for client-side components and one for sandbox components (the latter two exist as separate instances for every client connected to the server). Every environment needs its own runtime infrastructure. So we split the Plux.NET runtime core into a *Server Runtime*, a *Client Runtime* and a *Silverlight Runtime* each running in its own environment. Logically these nodes form a single entity represented by the *Runtime* component on the server (see Figure 7). For discovering extensions on the server and on the client the discovery mechanism also had to be split into a *Server Discovery* and a *Client Discovery* component, each monitoring local extension directories.

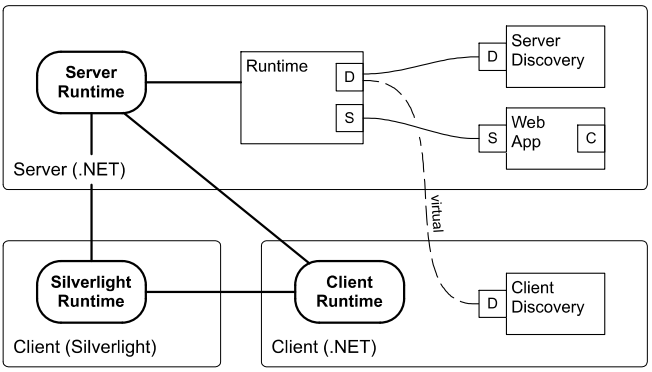


Figure 7: Runtime infrastructure and discovery distributed over several environments

As long as components from the same environment are plugged together everything is like in the rich client case: the local runtime raises a *Plugged* event to which the host component reacts by integrating the extension component. The interesting new feature is *virtual plugging*, which is necessary when the host and the extension live in different environments. If an extension E from environment Env_E should be plugged into a host H from environment Env_H proxies have to be generated on both sides. A proxy H_P representing host H is created in Env_E, and a proxy E_P representing extension E is created in Env_H (Figure 8). These proxies carry the same metadata as the components for which they stand so they can be plugged into matching slots like any other component. If H wants to call a method of E it does the call to the local proxy E_P, which uses the local runtime infrastructure to marshal the call and send a message to proxy H_P in the other environment. H_P then does the actual call to E. Any results are sent back in the same way. Thus the communication between components—from the same or from different environments—is completely transparent to the component developer.

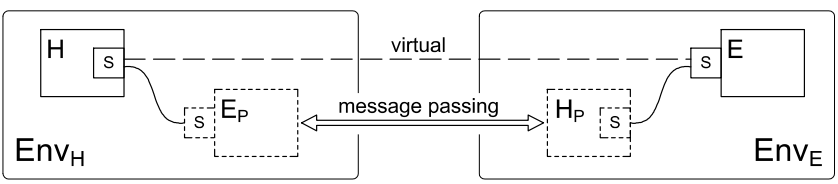


Figure 8: Virtually plugged components and their communication via proxies

Whenever a new extension is discovered on the server or on the client the local discovery component sends a broadcast to the runtime nodes of all other environments. The local runtime nodes decide whether proxies have to be generated. In the case of Silverlight extensions that were discovered on the server the *Silverlight Runtime* on the client requests a transfer of the extension from the server to the sandbox environment on the client.

It is also worth noting that all environments have a consistent copy of the web application's composition state, i.e. they know, which components the application consists of and which plugs are connected to which slots. So the runtime node of every environment can easily find out into which slots a new extension fits and whether the extension has to be plugged in locally or virtually through proxy components.

Since the runtime core is distributed, the individual runtime nodes have to use a communication channel for exchanging the components' metadata and their composition state. Additionally, the runtime core provides the communication infrastructure for the proxies of virtual plugged components. This infrastructure is based on the Windows Communication Foundation (WCF) API [Ca07]. As WCF supports several communication standards for distributed computing, the actually used communication technology can be configured by administrators. In our case study we used SOAP [GHM07] in combination with HTTP.

Since web applications can be used by many clients simultaneously, the runtime core has to deal with several composition states in parallel. In doing so it has to make sure not to run out of memory. Hence, it applies a well-known solution for this problem: Once the server has only little memory left, server-side components get released at the end of a request and are recreated for new requests. This approach ensures that web applications can scale up to serve many simultaneous requests without running out of server memory. This concept also enables the usage of server farms. The drawback is that composition and component states need to be persisted during successive requests. The composition state is persisted automatically while the component state has to be persisted by the components themselves using the infrastructure of the runtime core. Components can either use custom .NET attributes to declare which values should be persisted and restored, or they can react to automatically raised events for persisting and restoring.

The runtime does not only persist the server-side state, but also the states of the client environments. Thus, no matter from which computer a client connects to the application, it will always get the same state as it had last time, provided that possibly used client-side components are available on each computer.

4. Related work

Plug-in frameworks have become quite popular recently. However, most of them are either targeting rich-client applications, or have no dynamic composition support, or aim at web applications, but cannot be customized and extended by end users.

Eclipse [Ec03] is probably the most prominent plug-in platform today. It is written in Java and since version 3.0 it is based on the OSGi framework *Equinox* [Eq09]. Like Plux.NET it consists of a thin core and a set of plug-ins that provide further functionality. The major differences between Eclipse and Plux.NET are the following: Eclipse declares the metadata of plug-ins in XML files while Plux.NET declares them directly in the code using .NET attributes. Eclipse supports only rich client applications while our approach targets also multi-client web applications with extensions both on the server and on the client. Most importantly, the composition of Eclipse plug-ins has to be done manually, i.e. the host component has to use API calls to discover plug-ins, read their metadata and integrate them. In Plux.NET, plug-ins are discovered automatically and all matching slots are immediately notified by the runtime core, thus automating a substantial amount of composition work. Although Eclipse allows plug-ins to be added dynamically, the code for integrating plug-ins at startup time and at run time is different, whereas Plux.NET uses the same uniform mechanism for both cases.

Many component-based web applications are based on the *Java Enterprise Edition* (Java EE) [Su09]. Java EE is a software architecture that allows the development of distributed, component-based, multi-tier software running on an application server. Java EE applications are generally considered to be three-tiered where components can be installed on the Java EE server machine, the database machines, or the client machines. Web components are usually servlets or dynamic web pages (created with JavaServer Faces or Java Server Pages) running on the server, but they can also be defined as application clients or applets running on the client. Even though Java EE provides a framework for building component-based and distributed web applications it provides no automatic composition support for components. Composition has to be done programmatically. Moreover, Java EE does not provide an individual composition state for every end user, so users cannot customize or extend web applications for their special needs.

A further component system for distributed, component-based applications is *SOFA 2* [HP06, BHP06, BHP07]. It implements a hierarchical component model with primitive and composite components. Primitive components consist of plain code whereas composite components consist of other subcomponents. SOFA 2 has a distributed runtime environment which automatically generates connectors to support a transparent distribution of applications to different runtime environments. For communication the connectors use method invocation, message passing, streaming, or distributed shared memory. SOFA 2 allows dynamic reconfiguration of applications by adding and removing components as well as dynamic update of components at run time. In contrast to Plux.NET, however, SOFA 2 needs an ADL (Architecture Description Language) for describing the composition of components. Plux.NET does not need such a specification; its runtime composes an application on the fly using the declarations of slots and plugs provided by the Plux.NET components. We argue that this concept is more flexible and easier to maintain than a global ADL specification. Finally, browser-based web applications are not supported by SOFA 2. It has no multi-user support for individual composition states and no mechanism for persisting and restoring the composition states at run time.

Currently, the only way how end users can extend a web application is through client-based plug-in systems such as *Mozilla Firefox* [Mo09]. However, client plug-ins are not integrated into web applications. They only add usability features to user interfaces or enable web browsers to use advanced web technologies such as Flash, Silverlight, or Java Applets.

5. Summary and future work

In this paper we presented a dynamic plug-in framework for rich client and web applications with the focus on multi-user web applications that are extensible by end users. Each client can install its individual set of components and has its personal composition state. Components can be instantiated in different environments and on different computers, but are transparently composed into a single web application. Components for business logic can stay on the server, components that are connected to a client's local resources can be executed on the client-side, and user interface components can live in a client's rich internet environment such as Silverlight.

Since several aspects mentioned above have been realized only prototypically so far, we are still improving the distributed runtime core of Plux.NET. Furthermore, we are working on a security model for plug-ins, a layout manager for extensible component-based user interfaces, a keyboard shortcut manager and many other helpful tools for component-based software development.

References

[BHP06] Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, Seattle, USA, IEEE CS, ISBN 0-7695-2656-X, pp.40-48, August, 2006.

[BHP07] Bures, T., Hnetynka, P., Plasil, F., Klesnil, J., Kmoch, O., Kohan, T., Kotrc, P.: Runtime Support for Advanced Component Concepts, Proceedings of SERA 2007, Busan, Korea, IEEE CS, ISBN 0-7695-2867-8, pp. 337-345, August, 2007.

[Ca07] Chappell, D. (Microsoft): Introducing Windows Communication Foundation, http://msdn.microsoft.com/en-us/library/dd943056.aspx, September, 2007.

[Ec03] Eclipse Platform Technical Overview. Object Technology International, Inc., http://www.eclipse.org, February 2003.

[Eq09] Equinox Mission Statement., http://www.eclipse.org/equinox/, 2009.

[GHM07] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H., Karmarkar, A., Lafon, Y.: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), W3C Recommendation, April, 2007.

[HP06] Hnetynka, P., Plasil, F.: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models, Proceedings of CBSE 2006, Vasteras near Stockholm, Sweden, LNCS 4063, ISBN 3-540-35628-2, ISSN 0302-9743, pp. 352 - 359, (C) Springer-Verlag, June, 2006.

[Mo09] Mozilla Foundation: Firefox Browser, Free ways to customize your Internet, http://www.mozilla.com/en-US/firefox/personal.html, 2009

[MS08] Microsoft .NET Framework. http://www.microsoft.com/net/overview.aspx, 2008.

[MS09] Microsoft Silverlight, http://silverlight.net/, 2009.

[RWG09] Rabiser, R., Wolfinger, R., Grünbacher, P.: Three-level Customization of Software Products Using a Product Line Approach. 42nd Hawaii International Conference on System Sciences, HICSS-42, Big Island, Hawaii, USA, January, 5-8, 2009.

[Su09] Sun Microsystems, Inc.: The Java EE 6 Tutorial, Volume I, Basic Concepts Beta, http://java.sun.com/javaee/6/docs/tutorial/doc/JavaEETutorial.pdf, August, 2009.

[WDP06] Wolfinger, R., Dhungana, D., Prähofer, H., Mössenböck, H.: A Component Plug-in Architecture for the .NET Platform. Modular Programming Languages, Lightfoot, David; Szyperski, Clemens (Eds.), Lecture Notes in Computer Science , Vol. 4228, Proceedings of 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006.

[WRD08] Wolfinger, R., Reiter, S., Dhungana, D., Grünbacher, P., and Prähofer, H.: Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques. 7th IEEE International Conference on Composition-Based Software Systems, ICCBSS 2008, Madrid, Spain, February, 25-29, 2008.

[Wo08] Wolfinger, R.: Plug-in Architecture and Design Guidelines for Customizable Enterprise Applications, OOPSLA 2008 Doctoral Symposium, OOPSLA 2008, Nashville, Tennessee, October, 19-23, 2008.

[Wo10] Wolfinger, R.: Dynamic Application Composition with Plux.NET: Composition Model, Composition Infrastructure. PhD Thesis, Johannes Kepler University, Linz, Austria, 2009.

Representing Formal Component Models in OSGi*

Marco Müller, Moritz Balz, Michael Goedicke
Specification of Software Systems
Institute of Computer Science and Business Information Systems
University of Duisburg-Essen, Campus Essen, Essen, Germany
{marco.mueller, moritz.balz, michael.goedicke}@s3.uni-due.de

Abstract: Formal component models have been subject to research for decades, but current component frameworks hardly reflect their capabilities with respect to composition, dependency management and interaction modeling. Thus the frameworks don't exploit the benefits of formal component models like understandability and ease of maintenance, which are enabled when software is composed of hierarchical and reusable components that are loosely coupled, self-describing and self-contained. In this contribution, we try to examine the discrepancies between the state of research and the capabilities of an existing module framework, the widely-used OSGi bundle management framework for the Java platform. Based on this we propose modifications and enhancements to the OSGi framework that allow to exploit the benefits of formal component models in OSGi-based applications.

1 Motivation

When software becomes larger and more complex, modularity is needed. The breakdown of large systems into small, independent and manageable pieces promises to let systems be easier to understand, maintain and change. For this reason, component models are subject to research in information technology for almost forty years [Par72]. The component models that have been developed as a result do not only facilitate a structured creation of modular systems, but are in many cases also formally founded [AG97, CS01, CFGGR91, MDEK95, SG94]. The focus of such component models can be classified roughly into six areas: (1) Composition of components; (2) provision of functionality of a component and its appropriate description; (3) management of dependencies and the definition of required components; (4) instantiation of components; (5) modeling of interaction between components; (6) creation of executable systems by connecting component instances at deployment time. A formal foundation for these definitions allows for detailed specification of modules and their interaction, and also enables developers to verify desired characteristics of modular software systems.

However, while the research on this topic is thus very advanced, implementations of the related concepts are hard to find in current programming languages, platforms and frame-

*Part of the work reported herein was funded by the German Federal Ministry of Education and Research under the grant number 03FPB00320.

works. When modern object-oriented programming languages like Java or C# are considered, module definitions are optional at all and only provided by external frameworks. In addition, the features provided by these frameworks lack important parts of the expressiveness formal component models proposed long ago. Since the ability to partition large software systems into manageable pieces is still desirable, we will consider the widely-used OSGi [OSG05] framework, which is based on the Java programming language and platform, in this contribution. OSGi does not focus on components, but manages so-called *bundles* containing program code and libraries as well as their dependencies. On top of this, services can be defined, which provide implementations of interfaces across bundles and enable loose coupling.

We examine how far OSGi is appropriate to create module-based systems with respect to the features defined by formal component models. Then we will propose modifications and enhancements to OSGi, that must be made in order to reach this objective. First we summarize the features of formal component models in section 2. Based on this, we examine OSGi with respect to these features in section 3, leading to a comparison of desirable and existing functionality in OSGi. A proposal of changes to support the objectives of formal component models in OSGi is made in section 4, before we present an overview of related work in section 5 and conclude in section 6.

2 Formal Component Models

Different formal component models exist that focus on various aspects, including component interconnection [AG97], message flow [CS01], data abstraction and concurrency [CFGGR91], dynamic architectures [MDEK95, BHP06], modeling and prediction of non-functional attributes [GMRS08, RBH+07, Inf03]. In this section we will try to summarize the features of established component models to allow for an evaluation of OSGi in section 3. Considering these models, which all have different focuses, we can categorize their features into the following six areas mentioned in the introduction.

2.1 Composition

Composition of components considers the fact that modular software systems may be structured hierarchically. This is for example reflected in the component model proposed by Cox & Song [CS01] that distinguishes between simple and composite components. A simple component consists of *inports* and *outports* being sets of attributes defining provided and required functionality, a *function* describing the actual computations of the component and events that *trigger* the component to act. Composite components have no functions or triggers, but instead consist of *embedded component*s and a set of *connections* defining the interconnection of embedded components and the in- and outports of the composite component with the corresponding counterpart of an embedded component.

Enabling architects to construct composite components is stated to be the primary purpose

of the configuration language Darwin [MDEK95]. Simple components are essentially described by their provided and required services. Composite components contain instances of other components and specify the bindings between them. Internal components and their dependencies are hidden from external components, so that the compositions form a hierarchical abstraction. In general, components can be differentiated by their compositionality. Primitive components directly implement the functionality they provide, while composite components consist of components themselves, thus hiding the context from their enclosed components and vice versa.

2.2 Provided Interfaces

In larger systems, components offer services that can be used by other components. Since the context is not known beforehand and maybe not intended by the component developers, the provided interfaces must be described thoroughly. This is considered by Allen & Garlan who define component interconnection with so-called *connectors* [AG97] providing a formal basis to describe their behavior. In this case components define so-called *ports*, which are essentially state charts with messages, to describe exported behaviour, called *processes*. The messages between those processes contain untyped data.

Similarly, the Π language [CFGGR91] defines the so-called *export* of a component in three views: the *type view* describing exported data types, the *imperative view* describing the exported behaviour in an imperative manner, and the *concurrency view* describing concurrency constraints for the execution of the imperative operations. In summary, the export of a component is commonly described by a set of operations which can be called by the context of the component. However, the definitions vary in the degree of detail, especially regarding data types.

2.3 Dependencies

The provision of services introduces dependencies which must be managed during development and assembly of a component-based software. Since dependencies may be based not only on direct connections, but also on functional requirements that can be satisfied by different components, the requirements must be described precisely. An example for this is Darwin describing required services with an interface name, a communication mechanism and a data type. Data type and communication mechanism are not specific to the language, but interpretation of these is left to the underlying platform. Thus the dependencies are essentially the name of an interface. The implementation of a component can use the required service description without knowing the name of the component bound to this dependency at run time. The components are thus context-independent.

The dependency of a component in Π is called *import* and also defined in three views, similar to the export. The *type view* describes the imported data types with their operations, the *imperative view* describes the imported behaviour in an imperative manner, and the

concurrency view describes constraints for parallel execution of the expressed behaviour. The import is also context-independent, i.e. the implementation of the component can use the import without knowledge about the context of the component at run time.

Palladio [RBH+07], SOFA 2 [BHP06], and ROBOCOP [Inf03] use a different approach: Dependencies are also well-defined interfaces, but the interfaces are first-class entities, which are shared by the consuming and the providing component. Thus the components are not context-independent at design time, because the shared interface must be known.

Since dependencies require certain functionality, components must provide a description of the functionality they expect. Differences exist regarding the degree of details: The data types may be completely defined as in Π or be completely left to a surrounding platform as in Darwin. Additional properties like concurrency constraints are also supported by some models. For context independence, the requirements must be described locally to the component and thus without knowledge of its future context.

2.4 Instantiation

At run time multiple instances of components can exist. For this reason, in SOFA 2 a system is described as a component architecture. The components are instantiated and provided with a unique name for each component. The component instances can then be referenced in the architecture by connections. This expressiveness cannot be taken for granted, as it is for example not available in Cox & Song's model, where components are executed without considering instances. Nevertheless, in this case a component equivalence relation is defined to compare two components with each other. Components are divided into component classes where the members of a component class are syntactically identical and differ only by their identification. While the approaches are different, the need is clear to consider instances at least, if only in the case that singleton instances are equipped with identification mechanisms.

2.5 Interactions

In a system consisting of dependent components, the components must be able to communicate. This dynamic aspect of component interaction is considered by Allen & Garlan in the form that components contain processes that are subject to communication events. The communication events are locally defined in each component and connector specification. These context-independent specifications are glued together by the connector instances mapping the events and data parameters of one role to the counterparts of the second role.

The interfaces in Palladio optionally include a protocol definition, e.g. declared as a finite state machine or a regular expression. The protocol defines the sequence of operation calls, a client may call on a provided interface. In case of a required interface, the protocol describes how the client uses the required interface. Components in Π use a likewise approach for defining a sequence and concurrency constraints for operation calls for re-

quired and provided behaviour and types using so-called *path expressions* over operation names. In summary, since component communication is inevitable for a modular system, the related interaction is subject to specification by the formal component models.

2.6 Assembly

An executable system in the model of Allen & Garlan is constructed by the definition and instantiation of components and connectors as well as the definition of the interconnection between those elements. In these interconnections the ports of the component instances are bound to the roles of the connector instances. Darwin describes the system with a component definition instantiating embedded components and connecting their provided and required services directly. Commonly, a system is constructed by instantiating the desired components and interconnecting their provided services and dependencies using a separate configuration. This assembly must be considered a separate stage in the development process which can be taken only if the components themselves are finished, but before the system is being executed.

3 Components in OSGi

OSGi is a component system for the Java platform. It is widely-used, from embedded systems to server-based enterprise applications, since no adequate functionality is provided by the language or the platform. The basic functionality of OSGi is the management of so-called *bundles* which are technically Java libraries (Java Archive; JAR) containing compiled classes and binary resources. Bundles are configured in the descriptor file of their JAR file (`MANIFEST.MF`) with name-value pairs. The meta data include information about the ID, name, and version of a bundle. It also describes the relations to other bundles, either with a specification of bundle IDs, or with a definition of packages provided by other bundles. At run time, the OSGi framework is responsible for loading the bundles, resolving dependencies, and controlling the access to provided packages.

While this simple structure is an appropriate solution for dependency management and access control, it does not describe provided functionality of bundles in detail and forces a tight integration between bundles at the same time. Addressing these issues, an additional layer has been added to OSGi that supplements the bundle concept: The *Service Layer* is responsible for managing so-called *service components*. They encapsulate the functionality provided by a bundle by offering a named service, which is published in a registry and can be retrieved and accessed by any class running inside the OSGi framework. Their interface is described with a Java interface, thus using the Java semantics for method signatures and data types. Considering both, bundles and services, we will now relate the features of OSGi to that of the formal component models introduced above.

3.1 Composition

OSGi does not support composition of components. Bundles are only connected by their dependencies and must thus be considered simple components. The Service Layer does not add any additional semantics regarding composition since it only describes interfaces of bundles, but does not provide another view on bundles and their dependencies.

3.2 Provided Interfaces

Since OSGi allows to modularize applications, single bundles are likely to provide functionality to other bundles. They *export* packages containing Java types to make them available for use by other bundles. With exported and non-exported packages information hiding can be realized. Services allow for more encapsulation by offering just an interface description being published under a certain name, with the bundle itself instantiating and managing the underlying objects. This means that the description of provided functionality is possible without causing the need to reveal internal functionality of the bundle.

However, the semantics of interfaces and data types in use are simply those of Java. This causes a tight integration into the underlying Java platform. In addition, the bundles themselves are tightly integrated, since all type definitions being exported must be available to all using bundles in a shared bundle. This enforcement of direct dependencies contradicts on the one hand the principles of the formal component models which assume that components can describe their services completely independent, as for example defined in Π. On the other hand, the Java platform and a network of dependent bundles can provide a rich ecosystem of types which are hard to describe from scratch for each provided interface.

3.3 Dependencies

OSGi allows to specify dependencies between bundles in three ways: (1) By referencing the bundle name; in this case, all exported package of the dependency are available in the bundle. (2) By referencing package names; when bundles are available that export these packages, they are accessed. (3) By referencing service component descriptors; the reference is described by the Java interface, the cardinality, and whether the reference is optional. In all cases, the dependencies are resolved at run time when the bundles started, thus causing errors if dependencies cannot be satisfied.

This contradicts the specifications of formal component models since it leads to a tight integration between bundles. As described for the interface provision above, all types in use must be provided by bundles that are shared between all using bundles. When the bundle providing a service also exports the related types, the depending bundle is statically bound to it as illustrated at the left hand of figure 1. Since the service interface must be available for consumers at compile time, the interface has to be copied to implement different providing bundles for one service. A change in the service interface results in

incompatible services so that each consumer must be updated and recompiled.

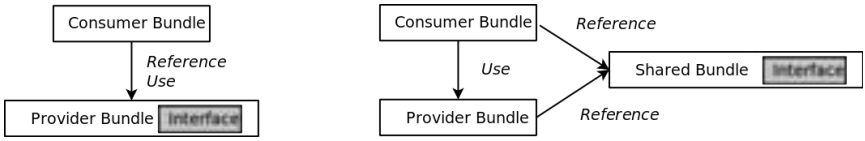


Figure 1: Independence of service components. At the left hand, the interface is packaged with the providing bundle, so that the consumer references the service bundle directly. At the right hand, the bundles are decoupled, allowing services to be provided with the same interface by multiple bundles.

These problems can be circumvented in OSGi by providing the service interface in a separate bundle which is referenced by all providers and consumers as shown at the right hand in figure 1. This has the advantage that consumers can be implemented without knowledge about the actual service provider and thus without a static reference to it. Different services can also be provided under the same interface, since only the shared type definitions must be known before the application is assembled. However, the interface still needs to be defined before the service consumer can be developed and each change on the interface results in the requirement to recompile the affected consumers.

In contrast, formal component models propose to define dependencies by not using shared types, but instead defining the functionality to import locally. This means that components describe all interfaces and related data types completely by themselves and are thus independent from external descriptions. At assembly time, these descriptions are sophisticated enough to to determine if the provided interface of any available component matches the requirements of the depending component. By this means, components are defined completely context-independently, which cannot be realized with OSGi.

3.4 Instantiation

OSGi bundles cannot be instantiated since they are simply collections of classes. Service components are instantiated by their owning bundles and registered at the service registry using a unique name, so that only one instance exists. More than one instance of a service component can only exist when different names for the instances are used, which could be considered confusing. The types that are exported in packages can be instantiated by consumers without restriction and independently from the OSGi framework. In contrast to the instantiation features of formal component models, OSGi does therefore not provide components with identification mechanisms but only with names for the single instances.

3.5 Interactions

As bundles mainly manage visibility of packages, the interaction between bundles is effectively arbitrary communication between Java types and thus not under control of the

framework. In the Service Layer, creation of a reference to service objects is controlled by the framework during lookup. However, the communication itself is not surveyed or intercepted by the OSGi framework. The features provided by some formal component models for specifying constraints, for example with respect to data types, value ranges, or concurrent access for component interaction, like path expressions in Π, are not realized.

3.6 Assembly

A software system consisting of OSGi bundles is assembled at run time. The wiring of services is controlled programmatically or descriptively in service component definitions. However, semantic validation is not possible before the system is started. The reason is that dependencies are described with names and Java types rather than independent interface and interaction descriptions, as are provided by formal component models. Thus required types and services must be available when a bundle is started, otherwise errors occur. A validation before bundles are started is not supported by OSGi since the availability of requirements cannot always be determined completely without activating the bundles.

4 Proposal for Improving of the OSGi Service Layer

We have seen that some OSGi features can be related to concepts of formal component models, especially on the Service Layer. We now propose a set of changes to OSGi in order to make use of formal component specifications. First we change the definition of dependencies and refine the description of provided services to create context-independent components. Second we introduce composite components to the Service Layer to enable component hierarchies. At last we introduce modeling of component interaction with path expressions and type conditions.

4.1 Dependencies

OSGi relies on the semantics of Java interfaces for services and on the Java platform for related data types. There are three ways to define dependencies and provision of data types that are not offered by the platform: First, shared data types can be defined in an own bundle which is accessed by all components participating in the communication. This is essentially extending the platform with the needed data types. Second, data types are defined by Java interfaces by the providing component. Their implementations are instantiated and managed by the provider and only accessed by the consumer.

Third, service interface and data types are completely specified by provider and consumer. The OSGi framework is responsible for resolving dependencies by matching both interfaces and data types at assembly time. At run time, the framework must map any data that is exchanged between both representations. This approach makes bundles and services

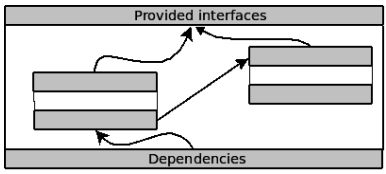


Figure 2: A composite component declaring composite provided resources and dependencies. The embedded components are hidden to the context.

context-independent since they are self-contained and do not rely on external specifications of data types. Tools can be developed that support assembly time validation as well as the run time connection and thus support developing context-independent components.

4.2 Composition

Composite components in formal component models hide subcomponents and cannot be distinguished from simple components from outside. Internally, their functionality relies on dependencies between components as illustrated in figure 2. To enable this behaviour in OSGi, the framework must consider libraries embedded in a bundle as bundles too, i.e. scan for component descriptors and activate bundles and services. Component descriptors of composite bundles are responsible for interconnecting subcomponents, defining composite export and import, and mapping them to exports and imports of subcomponents.

4.3 Interaction

Formal component models like Π and Palladio describe component interaction and take concurrency into account. We propose to add constraints regarding data types, value ranges and sequences of calls to exported and imported services.

Data types described in OSGi are limited to Java semantics. While the static type system is already expressive, it does e.g. not allow to specify constraints to method parameters apart from the built-in data type constraints. If, for example, a parameter of an operation declared in a component's exported behaviour must not be null, this requirement could only be stated in a documentation. To describe value ranges for data types, we propose to consider approaches like the Java Modeling Language [BHS07] and use the related concepts in service component definitions. Data types and behaviour are annotated with pre- and postconditions using XML descriptor files or Java annotations. The platform that controls communication between bundles and services can evaluate whether the conditions are met at run time, which is illustrated in figure 3, and prevent any invalid invocations.

The sequence of calls of exported and imported services can be described statically using Π's path expressions. These could e.g. be used to enforce the construction of a type before

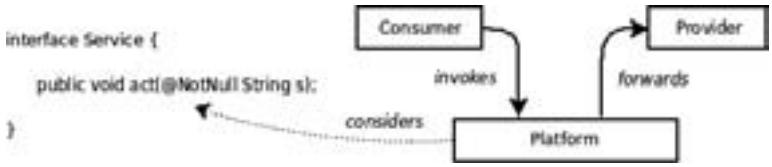


Figure 3: The proposal to use type constraints when the OSGi platform forwards requests between services. For example, meta data in the interface program code can force parameters to be not null, which is validated by the platform during requests.

it is used. Path expressions in this context consist of operation names and operators describing the permitted call sequence. For example, the sequence `create;(*[{read}|write]*);destroy` describes that first the operation *create* must be called. Afterwards optionally and repeatedly either *concurrent* `read` calls or a single *write* call can be executed. At last the operation *destroy* must be called. The framework has to verify that the communication complies with the path expressions and prevent it if the constraints are not met. In the future, one can imagine tools that use static code analysis to verify call sequences already at development time based on these interface descriptions.

4.4 Assembly

When independently-developed components are assembled in order to create an executable system, their requirements regarding dependencies, services and data types must be verified. In OSGi this currently happens at run time when bundles are activated. We propose to create tools that assemble the software before it is started and at that time consider all descriptions of imports, exports, and communication constraints introduced above. This stage would allow for a more thorough verification of the interconnections between components, which is important since the context of component-based systems is not always known by component developers beforehand. Based on the same information, additional tools could monitor the current state and the interactions of the running system afterwards.

5 Related Work

Several approaches aim at providing module concepts inside Java. In the Spring framework so-called *Spring Beans* can be defined including dependencies on other beans. Spring Dynamic Modules (Spring DM) [CHLP08] enables developers to use Spring Beans as OSGi Services and vice versa. Spring DM provides these beans to the system using Java interfaces. However, components in this context do not differ from the OSGi Service Layer. The Java Enterprise Edition allows to write distributed server-side applications which also consist of modules. In this context, modules are called *Session Beans* [Sun08] and are able to define dependencies using Java interfaces and annotations. However, they are tightly coupled to their type definitions, so that context independence is not possi-

ble. Modeling of interfaces and interactions also does not exceed the functionality of the Java language. In the CORBA middleware, the CORBA Component Model [Obj06] uses loosely-coupled components with parameterized events as interaction mechanism. Interfaces and data types are defined independently from implementation languages in an Interface Definition Language providing more detailed features regarding value ranges and parameter constraints than Java. However, this is not well integrated in Java because no formal mapping between the language elements exists and generated helper classes are used for data conversion. Beanome [CH02] manages components in OSGi bundles that are described by XML files. However, this approach uses shared interfaces, rendering the components context-dependent. Beanome was developed for the outdated OSGi 2 and is not maintained anymore.

6 Conclusion

In this contribution we examined a selection of formal component models and a widely-used practically driven component framework in Java, based on the observation that concepts of modularity are widely accepted, but hardly taken to full advantage in current object-oriented programming languages and frameworks. We compared the features of the formal component models with the component framework OSGi. Dependencies between bundles and services in OSGi can be related to component concepts, but do not allow a loose coupling. Composite components, descriptions of exported interfaces apart from Java interfaces, and modeling of interactions are not supported in OSGi at all.

Based on this, we proposed enhancements to OSGi with respect to the description of provided interfaces and services, loose coupling by means of more precise definitions of required interfaces, and application of path expressions to OSGi for describing interactions. In future work we plan to build a prototype of the enhanced OSGi framework, and tools that support the development of single components as well as the assembly of complete software systems from single components using the proposed mechanisms. We also plan to consider more formal component models and more frameworks for comparison, to find out how practically driven frameworks can be used for differently focused architectural views. With an implementation of these concepts we hope to reduce the gap between research and practice by introducing advanced concepts into the widely-accepted OSGi framework and thus supporting the idea of component-based development.

References

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlag New York, Inc., 2007.

[CFGGR91] Joachim Cramer, Werner Fey, Michael Goedicke, and Martin Große-Rhode. Towards a Formally Based Component Description Language. In *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, pages 358–378, London, UK, 1991. Springer-Verlag.

[CH02] Humberto Cervantes and Richard S. Hall. Beanome: A Component Model for the OSGi Framework. In *Software Infrastructures for Component-Based Applications on Consumer Devices*, 2002.

[CHLP08] Adrian M Colyer, Hal Hildebrand, Costin Leau, and Andy Piper. Spring Dynamic Modules Reference Guide, 1.2.0, 2008. `http://static.springframework.org/osgi/docs/1.2.0/reference/pdf/spring-dm-reference.pdf`.

[CS01] Philip T. Cox and Baoming Song. A Formal Model for Component-Based Software. In *Proc. IEEE Symposia on Human Centric Computing Languages and Environments*, 2001.

[GMRS08] Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo, and Antonino Sabetta. KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. In *The Common Component Modeling Example: Comparing Software Component Models*, pages 327–356, Berlin, Heidelberg, 2008. Springer-Verlag.

[Inf03] Information Technology for European Advancement. ROBOCOP: Robust Open Component Based Software Architecture for Configurable Devices Project, Deliverable 1.5 - Revised specification of framework and models, July 2003. http://www.hitech-projects.com/euprojects/robocop/deliverables.htm, visited at 2009-12-02.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. *Lecture Notes in Computer Science*, 989:137–153, 1995.

[Obj06] Object Management Group, Inc. CORBA Component Model Specification, Version 4, April 2006. `http://www.omg.org/cgi-bin/doc?formal/06-04-01`.

[OSG05] OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.1*. IOS Press, Inc., 2005.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[RBH+07] Ralf Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical report, Chair for Software Design & Quality (SDQ), University of Karlsruhe (TH), Germany, May 2007.

[SG94] Harald Schumann and Michael Goedicke. Component-Oriented Software Development with PI. Technical Report 1/94, Specification of Software Systems, Department of Mathematics and Computer Science, University of Essen, 1994.

[Sun08] Sun Microsystems, Inc. JSR 318: Enterprise JavaBeans™3.1 - Proposed Final Draft, March 2008. `http://jcp.org/en/jsr/detail?id=318`.

Automated Benchmarking of Java APIs

Michael Kuperberg[1], Fouad Omri[1], Ralf Reussner[1]

[1]Chair Software Design & Quality, Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
{michael.kuperberg|fouad.omri|ralf.reussner}@kit.edu

Abstract: Performance is an extra-functional property of software systems which is often critical for achieving sufficient scalability or efficient resource utilisation. As many applications are built using application programmer interfaces (APIs) of execution platforms and external components, the performance of the used API implementations has a strong impact on the performance of the application itself. Yet the sheer size and complexity of today's APIs make it hard to manually benchmark them, while many semantical constraints and requirements (on method parameters, etc.) make it complicated to automate the creation of API benchmarks. Additionally, modern execution platforms such as the Java Virtual Machine perform extensive nondeterministic runtime optimisations, which need to be considered and quantified for realistic benchmarking. In this paper, we present an automated solution for benchmarking any large APIs that are written in the Java programming language, not just the Java Platform API. Our implementation induces the optimisations of the Just-In-Time compiler to obtain realistic benchmarking results. We evaluate the approach on a large subset of the Java Platform API exposed by the base libraries of the Java Virtual Machine.

1 Introduction

Performance (e.g. response time) is an important extra-functional property of software systems, and is one of the key properties perceived by the users. Performance influences further software qualities, such as scalability or efficiency of resource usage. Addressing the performance of an application should not be postponed to the end of the implementation phase, because the cost of fixing performance issues increases as the application grows. While architecture-based performance prediction approaches such as Palladio [BKR09] exist, the factual performance is determined by the implementation of an application.

Therefore, software engineers should address software performance during the entire implementation phase. As many applications are built using application programmer interfaces (APIs) of execution platforms and external components, the performance of these APIs has a strong impact on the performance of the application itself. When an application itself offers APIs, their performance has to be benchmarked and controlled as well.

While profiling tools such as VTune [Int09] help with finding performance issues and "hot spots", they are not suitable for performance testing of entire APIs created by software engineers. Additionally, many applications target platform-independent environments such as Java Virtual Machine (JVM) or .NET Common Language Runtime (CLR), which offer large application programming interfaces (APIs) and the corresponding API implemen-

taion. A given functionality is often provided by several alternative API methods, but there is no performance specification to help choose between them. Altogether, benchmarking both the required and the provided APIs completely by hand is an unrealistic task: for example, the Java platform API has several thousands of methods.

Thus, to benchmark methods of APIs, developers and researchers often manually create *microbenchmarks* that cover only tiny portions of the APIs (e.g. 30 "popular" methods [ZS00]). Also, the statistical impact of measurements error is ignored and the developers must manually adapt their (micro)benchmarks when the API changes. Additionally, benchmarking API methods to quantify their performance is a task that is hard to automate because of many semantical constraints and requirements (on method parameters, type inheritance, etc.). To obtain realistic results, extensive runtime optimisations such as Just-in-Time compilation (JIT) that are provided by the JVM and the CLR need to be induced during benchmarking and quantified. Thus, there exists no standard automated API benchmarking tool or strategy, even for a particular language such as Java.

The contribution of this paper is an automated, modular solution to create benchmarks for implementations of very large black-box APIs, with a focus on challenges arising for APIs that are written in Java. Our solution is implemented to induce the optimisations of the Java JIT compiler, and quantifies its effects on the performance of benchmarked methods. The execution of benchmarks is also automated, and when the API or its implementation change, the benchmarks can be regenerated quickly, e.g. to be used for regression benchmarking. Our solution is called APIBENCHJ and it requires neither the source code of the API, nor a formal model of method input parameters. Also, in contrast to black-box *functional* testing, APIBENCHJ is not searching the parameter space for (unexpected) runtime exceptions and errors - instead, it uses existing techniques such as heuristic parameter generation [KOR09] to *avoid* errors and to find meaningful (legal and representative) method parameters for benchmarking.

We evaluate the presented framework by automatically benchmarking 1458 methods of the frequently-used, significant Java Platform API packages `java.util` and `java.lang`. Afterwards, we compare APIBENCHJ benchmarking results to results of fully manual benchmarking. In this context, we discuss how APIBENCHJ decreases the need for manual work during API benchmarking, and also how APIBENCHJ should be extended to better quantify parametric performance dependencies.

The remainder of the paper is organised as follows: in Sec. 2, we outline the foundations of our approach. In Sec. 3, related work is presented and compared to APIBENCHJ. An overview of APIBENCHJ is given in Sec. 4, while the details of the implementation are described in Sec. 5. Sec. 6 describes our evaluation. In Sec. 7, we outline the assumptions and limitations of our approach. The paper concludes with Sec. 8.

2 Foundations

The implementation of an API method in Java is usually provided by a library, i.e. as bytecode of Java classes (we do not consider native methods or web services here). Benchmarking a method means systematically measuring its response time as it is executed.

To execute a method, it must be called by some custom-written Java class, i.e. the bytecode of such a suitable caller class must be loaded and executed by the JVM (in addition to the callee bytecode). There are three different techniques for caller construction: (1) using the *Java Reflection API* to dynamically call methods at runtime, (2) using *code generation* to create caller source code that is compiled to executable caller classes, and (3) using *bytecode engineering* techniques to directly construct the binary Java classes that call the benchmarked methods. All these three techniques need to be examined with respect to their impact on the behaviour of the JVM's (JIT just-in-time compilation, etc.) and on the measurement itself (e.g. overhead of Java Reflection API usage).

For example, the constant folding algorithm implemented by JIT can identify simplification possibilities by replacing successive calls to an arithmetic operation which use the same parameters by a constant node in the dependency graph of the JIT compiler [CP95]. In order to avoid constant folding during benchmarking, the JIT compiler should not identify input parameters of the benchmarked methods as constants. Also, he measurements have to be carried out with respect to statistical validity, which is influenced by the resolution of the used timer and the duration of the benchmarked method. Such challenges have to be met in order to avoid misleading benchmarking results.

During benchmarking, in order to execute a method that has one or several input parameters, these parameters *must* be supplied by the caller and they must fulfill implicit and explicit requirements. For example, a `IndexOutOfBoundsException` is thrown for `String.substring(int beginIndex)` if `beginIndex < 0` or if `beginIndex > string.length()` since these parameters are inappropriate. In general, method parameters can be of several types: primitive types (`int`, `long` etc.), object types that are 'boxed' versions of primitive types (e.g. `Integer`), array types and finally of general object or interface types.

For primitive parameter types, often only specific values are accepted (cf. `String.substring` method above), and if a 'wrong' parameter value is used, the invoked method will throw a runtime exception. Very often, such exceptions do not appear in method signatures, and are also undocumented in the API documentation. Even for this single integer parameter, randomly guessing a value (until no runtime exception is thrown) is not recommended: the parameter can assume 2^{32} different values.

For parameters of types extending `java.lang.Object`, additional challenges arise [KOR09]. Unfortunately, almost all APIs provide no formal specification of parameter value information, and also provide no suitable (functional) test suites or annotations from which parameters suitable for benchmarking could be extracted.

In our previous work [KOR09], we have addressed the issue of automated parameter generation using novel heuristics, and have successfully evaluated it for several Java Platform API packages. By devising a modular approach, we have also proposed pluggable alternatives to these heuristics, e.g. recording parameters used in functional tests, or using values specified by humans. Therefore, in this paper, we concentrate on the actual generation, execution and evaluation of the API benchmarks, and assume the parameters as given.

An API can cover a vast range of functionalities, ranging from simple data operations and analysis up to network and database access, security-related settings, hardware ac-

cess, and even system settings. Hence, the first consideration in the context of automated benchmarking is to set the limits of what is admissible for automated benchmarking.

For example, an automated approach should be barred from benchmarking the method `java.lang.System.exit`, which shuts down the Java Virtual Machine. Likewise, benchmarking the Java Database Connectivity (JDBC) API would report the performance of accessed database, not the performance of the JDBC API, and it is likely to induce damage on database data. Thus, JDBC as part of the Java platform API is an example of an API part that should be excluded from automated benchmarking. APIBENCHJ handles exclusion using patterns that can be specified by its users. From the elements of an API that are *allowed* for automated benchmarking, the only two element types that can be executed and measured are non-abstract methods (both static and non-static) and constructors (which are represented in bytecode as special methods). Opposed to that, neither class fields nor interface methods (which are unimplemented) can be benchmarked.

3 Related Work

A considerable number of benchmarks for Java has been developed which differ in target JVM type: there are benchmarks for Java SE (e.g. [Cor08]), Java EE (e.g. [Sta08]), Java ME [Pie], etc. Java benchmarks also differ in scope (performance comparison of algorithms vs. performance comparison of a platform), size/complexity, coverage, and actuality; [BSW+00] provides an extensive but incomplete list.

While *comparative* benchmarking yields "performance *proportions*" or "performance *ordering*" of alternatives, our work needs to yield precise quantitative metrics (execution duration), parameterised over the input parameters of methods. Quantitative method benchmarking was done in HBench:Java [ZS00], where Zhang and Seltzer have selected and *manually* benchmarked only 30 API methods, but they did not consider or quantify the JITting of the JVM.

In the following, we do not discuss Java EE benchmarks because they target high-level functionality such as persistence, which is used transparently provided by the application servers using dependency injection, rather than direct API method invocations.

For Java SE, SPECjvm2008 [Cor08] is a popular benchmark suite for comparing the performance of Java SE Runtime Environments. It contains 10 small applications and benchmarks focusing on core Java functionality, yet the granularity of SPECjvm2008 is large in comparison to API methods benchmarking of the benchmark we present in this paper. Additionally, the Java Platform API coverage of SPECjvm2008 is unknown, and the performance of individual API methods cannot be derived from SPECjvm2008 results.

Other Java SE benchmarks such as Linpack [lin07] or SciMark [sci07] are concerned with performance of both numeric and non-numeric computational "kernels" such as Monte Carlo integration, or Sparse Matrix multiplication. Some Java SE benchmarks (e.g. from JavaWorld [Bel97]) focus on highlighting the differences between Java environments, determining the performance of high-level constructs such as loops, arithmetic operations, exception handling and so on. The UCSD Benchmarks for Java [GP] consist of a set of low-level benchmarks that examine exception throwing, thread switching and garbage

collection. All of these benchmarks have in common that they neither attempt to benchmark atomic API methods nor benchmark *any* API as a whole (most of them benchmark mathematical kernels or a few Java platform methods). Additionally, they do not consider runtime effects of JVM optimisations (e.g. JIT) systematically and they have not been designed to support non-comparative performance evaluation or prediction.

4 Overview of the APIBENCHJ Framework

In this section, we give a high-level overview of APIBENCHJ, while relevant details of its implementation are left to Sec. 5. The output for APIBENCHJ is a platform-independent suite of executable microbenchmarks for the considered API which runs on any Java SE JVM. Fig. 1 summarises the main steps of control flow in APIBENCHJ, and we explain it in the following (all steps are automated, and the user only needs to manually deploy the result of benchmark generation between steps 6 and 7). The steps highlighted with bold boxes form the focus of this paper, and are presented in more detail.

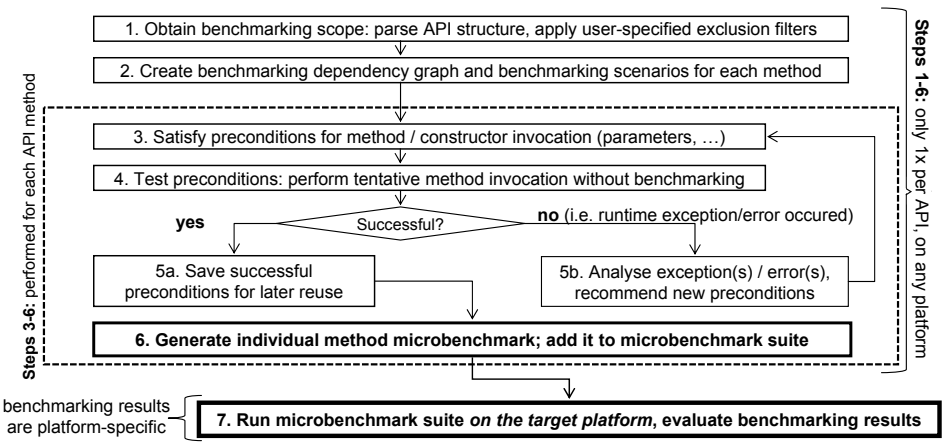


Figure 1: APIBENCHJ : overview of automated API benchmarking

Step 1 starts with *parsing and storing the API structure* to identify the relations between API elements, e.g. inheritance relations and package structure. APIBENCHJ can operate directly on bytecode and does not requires source code, i.e. it is suitable for black-box APIs whose implementation is not exposed. The Java platform and its Reflection API do not provide sufficient functionality for this task, e.g. one cannot programmatically retrieve all implementers of an interface. Thus, APIBENCHJ has its additional tools to parse the API structure using the bytecode classfiles of its implementation. Step 1 also applies *user-specified exclusion filters* to exclude entities that must not be benchmarked automatically, as described in Sec. 2. The exclusion filters are specified beforehand by users (i.e. APIBENCHJ does not try to exclude such entities itself). Filters can be package names, classes implementing a specific interface or extending a given class, etc.

Step 2 in Fig. 1 creates *benchmarking scenario(s)* for each method. Scenarios describe the *requirements* for benchmarking, e.g. which parameters are needed and which classes must

be instantiated *before* the considered method can be benchmarked. Actual runtime *values and objects* are created/instantiated later, in steps 3 through 7. In APIBENCHJ, a scenario consists of *preconditions*, the actual *benchmarked operation* and the *postconditions* for a method invocation. At the beginning, step 2 creates a *benchmarking dependency graph*, which holds relations such as "`String.contentEquals` must be preceded by initialisation of a `String` instance", or "the constructor `String()` has no preconditions". As several constructors for `String` and `StringBuffer` exist, several scenarios can be created which differ in the choice of constructors used to satisfy preconditions, and which allow the quantitative comparison of these choices. Step 2 can also compute metrics for the complexity of benchmarked methods (on the basis of their signature, which is fully visible in bytecode), so that step 3 can start with the methods having lowest complexity.

Step 3 starts with *trying to satisfy the precondition requirements* of a benchmarking scenario. Satisfying benchmarking requirements from Step 2 means generating appropriate method parameters, invocation targets, etc. A precondition may have its own preconditions, which APIBENCHJ must then satisfy first. As discussed in Sections 1 and 2 as well as in our previous work [KOR09], automation of these tasks is challenging due to runtime exceptions and the complexity of the Java type hierarchy/polymorphism. APIBENCHJ incorporates a combined approach to this challenge by providing a plug-in mechanism with different precondition sources which can be ranked by their usefulness. For example, *manual specification* has a higher rank than *heuristic search*, with *directed brute-force search* having the lowest ranking of the three. If, for example, APIBENCHJ finds that no manual plug-in exists for a precondition type, it could choose the heuristic search plug-in described in [KOR09]. The generated preconditions are likely to provoke runtime exceptions, as discussed in Sec. 2. Hence, they must be tested before being accepted.

Step 4 performs a *tentative method invocation* to test that using the generated preconditions does not lead to runtime exceptions (if such an exceptions occurs APIBENCHJ proceeds with **step 5b**). The error handler in step 5b triggers a new attempt to satisfy preconditions of the considered benchmarking scenario, or gives up the scenario if a repetition threshold is surpassed (this threshold serves to prevent infinite or overly long occupation with one scenario, especially if using brute-force parameter search).

Step 5a is entered if the tentative invocation succeeds, and the information on successful precondition values are internally saved for future reuse. The saved information may be a pointer to the successful heuristic, pointer to a code section that has been manually specified by a human, or a serialised parameter value. Due to space limitations, we do not describe the design rationale and the implementation of the mechanisms in step 5a/5b here but refer the reader to [KOR09].

Step 6 generates an executable microbenchmark for the considered scenario, using successfully tested precondition values. The generated microbenchmarks explicitly address measurement details such as timer resolution [KKR09b], JVM optimisations, etc. The *execution* of the resulting microbenchmark does not require the APIBENCHJ infrastructure that implements steps 1 through 6 - each microbenchmark is a portable Java class that forms a part of the final *microbenchmark suite*. The microbenchmark suite also includes additional infrastructure for collecting microbenchmark results and evaluating them. In Section 7, we discuss the parameter representability of the generated benchmarks.

5 Implementation of APIBENCHJ

In this section, we assume that appropriate method parameters are available (e.g. obtained from existing functional tests, human input, or heuristics outlined in [KOR09]. We also assume that the invocation targets for non-static methods (see steps 1-5 in Sec. 4) are available. Using the results of [KKR09b], we know the accuracy and invocation cost of the timer method used for measurements, and thus can compute the number of measurements needed for a given confidence level (see [Omr07] for details).

The remaining steps 6 (generating individual microbenchmarks) and 7 (executing the benchmarks) are discussed in this section. First, we discuss the runtime JVM optimisations and how they are addressed (Sec. 5.1), followed by the discussion in Sec. 5.2 on why bytecode engineering is used to construct the microbenchmarks.

5.1 JIT and other JVM Runtime Optimisations

Java bytecode is platform-independent, but it is executed using interpretation which is significantly slower than execution of equivalent native code. Therefore, modern JVMs monitor the execution of bytecode to find out which methods are executed frequently and are computationally intensive ("hot"), and optimise these methods.

The most significant optimisation is Just-in-Time compilation (JIT), which translates the hot method(s) into native methods *on the fly*, parallel to the running interpretation of the "hot" method(s). To make benchmarked methods "hot" and eligible for JITting, they must be executed a significant number of times (10,000 and more, depending on the JIT compiler), before the actual measurements start. JIT optimisations lead to speedups surpassing one order of magnitude [KKR08], and an automated benchmarking approach has to obtain measurements for the unoptimised and the optimised execution as both are relevant.

Different objectives lead to different JITting strategies, e.g. the Sun Microsystems Server JIT Compiler spends more initial effort on optimisations because it assumes long-running applications, while the Client JIT Compiler is geared towards faster startup times. We have observed that the Sun Server JIT Compiler performs multi-stage JITting, where a "hot" method may be repeatedly JIT-compiled to achieve even higher speedup if it is detected that the method is even "hotter" that originally judged.

Therefore, the benchmarks generated by APIBENCHJ can be configured with the *platform-specific* threshold number of executions ("warmup") after which a method is considered as "hot" and JITted by that platform's JIT compiler. To achieve it, we have implemented a calibrator which uses the `-XX:+PrintCompilation` JVM flag to find out a platform's calibration threshold, which is then passed to the generated benchmarks.

We must ensure that JIT does not "optimise away" the benchmarked operations, which it can do if a method call has no effect. To have *any* visible functional effect, a method either returns a value, changes the value(s) of its input parameter(s), or has side effects not visible in its signature. These effects can be either *deterministic* (same effect for the same combination of input parameters and the state of the invocation target in case of non-static methods) or *non-deterministic* (e.g. random number generation). If a method has non-deterministic effects, our approach simply has to record the effects of each method

invocation to ensure that the invocation is not optimised away, and can use rare and selective logging of these values to prevent JIT from "optimising away" the invocations. But if the method has deterministic effects, the same input parameters cannot be used repeatedly, because the JVM detects the determinism and can replace *all* the method invocation(s) directly with a *single* execution (native) code sequence, e.g. using "constant folding". This forms an additional challenge to be solved in APIBENCHJ.

Thus, we would need to supply different *and* performance-equivalent parameters to methods with deterministic behaviour, and we have solved this challenge by using array elements as input parameters. By referencing the `i`th element of the arguments array `arg` in a special way (`arg[i%arg.length]`), we are able to "outwit" the JIT compiler, and also can use arrays that are significantly shorter than the number of measurements. Altogether, this prevents the JIT compiler from applying constant folding, identity optimisation and global value numbering optimisations where we do not want them to happen.

Other JVM optimisations such as Garbage Collection interfere with measurements and the resulting outliers are detected by our implementation in the context of statistical evaluation and execution control.

5.2 Generating Executable Microbenchmarks

Using the Java Reflection API, it is possible to design a common flexible microbenchmark for all methods of the benchmarked API, provided that the latter are invoked with the Reflection API method `method.invoke(instanceObj, params)`. However, invoking benchmarked API methods dynamically with the Reflection API is very costly [FF04] and will significantly bias the measured performance.

Source code generation is the straightforward way to construct microbenchmarks, and it uses models (templates) that represent the source code of the microbenchmarks, where each microbenchmark is specific to a single method of the Java API. However, the manual generation of the models and code templates for each API method would be extremely work-intensive and would contradict the goal of the presented work which is to automate of the Java benchmarking. Consequently, the scope of the benchmark would be limited to specific Java implementations.

APIBENCHJ directly creates the 'skeleton' bytecode for a microbenchmark, using the Javassist bytecode instrumentation API [Chi]. This 'skeleton' contains timer method invocations (e.g. calls to `nanoTime()`) for measuring the execution durations. The 'skeleton' also contains control flow for a warmup phase which is needed to induce the JIT compilation (cf. Sec. 5.1). Thus, two benchmarking phases are performed: one for the 'cold' method (before JIT), and one for the hot (after JIT).

For each benchmarking scenario with appropriate preconditions, APIBENCHJ creates a dedicated microbenchmark that starts as a bytecode copy of the 'skeleton'. Then, the actual method invocations and preconditions are added to the 'skeleton' using Javassist instrumentation. Finally, APIBENCHJ renames the completed microbenchmark instance, so that each microbenchmark has a globally unique class name/class type, and all microbenchmarks can be loaded independently at runtime. An infrastructure to execute the microbenchmarks and to collect their results is also part of APIBENCHJ.

6 Evaluation

We have identified the following three metrics for evaluating APIBENCHJ :

- **Precision**: compare APIBENCHJ results to "best-effort"manual benchmarking
- **Effective coverage** of packages/classes/methods
- **Effort** (time and space) of microbenchmark generation and execution

Once the APIBENCHJ implementation will be complemented by a component to detect parametric performance dependencies, a fourth metric (detectability of linear parametric dependencies) can be added. Of course, detecting parametric performance dependencies requires enough input data (different parameter, different invocation targets). This aspect will be addressed in future work. All following measurements were performed on a computer with Intel Pentium 4 2.4 GHz CPU, 1.25 GB of main memory and Windows Vista OS running Sun JRE 1.6.0_03, in `-server` JVM mode.

6.1 Precision of Automated Benchmarking

To evaluate the precision of automated benchmarking performed by APIBENCHJ, we had to compare its results to results of manual benchmarks (which were not readily available and had to be created for the evaluation). This comparison is needed to evaluate the benchmark generation mechanism of APIBENCHJ; to enable a fair comparison, method parameters (and also method invocation targets) must be identical in both cases.

Hence, automated benchmarking was done first, and method preconditions during its execution were recorded and afterwards reused during manual benchmarking. This comparison is an indicator of whether the microbenchmark generation mechanism (cf. Sec. 4) generates microbenchmarks which will produce realistic results w.r.t JIT etc.

The method `java.lang.String.substring(int beginIndex, int endIndex)` was selected as a representative, because its performance is nontrivial and because its declaring class is used very often. This method was benchmarked with an invocation target `String` of length 14, `beginIndex` 4 and `endIndex` 8.

The result of manual "best-effort" benchmarking performed by a MSc student with profound knowledge of the JVM was 9 ns. The benchmarking result of APIBENCHJ (after removing GC-caused outliers) had the following distribution: 7 ns for 19% of measurements, 8 ns: 40%, 9 ns: 22.5%, 10 ns: 9%, 11 ns: 4%, and 12 ns for 5.5% of measurements. Thus, the average results of APIBENCHJ are 8.555ns, which constitutes a deviation of 5% compared to manual benchmarking. Note that a distribution and not just a single value is observed in APIBENCHJ because the JVM execution on a single-core machine is interrupted by the OS scheduler to allow the OS other applications to use the CPU.

Clearly, this is a promising result, but it does not give any guarantees for other parameter values of `substring`, or for other API methods. At the same time, we think that it is a strong argument for the generation mechanism described in Sec. 5.

6.2 Effective Coverage of Packages/Classes/Methods

APIBENCHJ can benchmark *all* the methods for which correct (*appropriate*) and *sufficient* input parameters (and invocations targets for non-static methods) are given. By *sufficient*, we mean that the benchmarking method can be executed repeatedly with the input parameters. For example, the `java.util.Stack` class contains the method *pop()* which should be benchmarked, which means that the method must be called several hundred times to account for timer resolution. If the `Stack` does not contain enough elements to call `pop`, an `EmptyStackException` is thrown - thus, the invocation target (the used Stack instance) must be sufficiently pre-filled.

If parameter generation is automated, the coverage is less than 100% because not all parameters are generated successfully. In [KOR09], we have evaluated the coverage of heuristic parameter finding, which we summarise in this subsection since APIBENCHJ can successfully generate and run benchmarks for all parameters that were found in [KOR09].

For the `java.util` package of the Java platform API, APIBENCHJ can thus benchmark 668 out of 738 public non-abstract methods, which is a success rate of 90.51%. Similarly, for the `java.lang` package, APIBENCHJ can benchmark 790 out of 861 public non-abstract methods, which is a success rate of 91.75%, with an effort comparable to `java.util`.

6.3 Effort

The benchmarking for `java.util` took 101 min due to extensive warmup for inducing JIT optimisations; the heuristic parameter generation for it took additional 6 minutes. The serialised (persisted) input parameters (incl. parameters to create invocation targets) together with persisted benchmarking results occupy 1148 MB on hard disk for the `java.util` package, but only 75 MB for the `java.lang` package.

The generation of microbenchmarks using bytecode engineering is very fast. For the `String` method `contains(CharSequence s)`, the generation of the microbenchmark took less than 10 ms. The actual benchmarking took ca. 5000 ms: APIBENCHJ repeated the microbenchmark until a predefined confidence interval of 0.95 was reached (which required 348 repetitions). The number of repetitions depends on occurrence of outliers and on the stability of measurements in general.

A comprehensive *comparison* of the total effort for benchmarking using manually created microbenchmarks and of benchmarking using APIBENCHJ is desirable. However, to get a reliable comparison, a controlled experiment needs to be set up according to scientific standards and this would go beyond the focus and the scope of the presented paper.

7 Assumptions and Limitations

In this paper, we assume that appropriate and representative method parameters are provided by existing heuristics [KOR09], manual specification, or recorded traces. As described in Sec. 2, automated benchmarking should not be applied to API parts which can produce a security issue or data loss. The decision to exclude such API parts from benchmarking can only be produced by a human, not by APIBENCHJ itself.

As interface methods and abstract methods have no implementation, they cannot be benchmarked directly. At runtime, one or several non-abstract type(s) that implements the interface are candidates to provides implementation(s) for the interface methods. Performance prediction thus requires that these candidate type(s) are considered - then, the results of the corresponding microbenchmark(s) can be used. If such runtime information cannot be obtained in a reliable way, the performance of invoking an interface method cannot be specified for performance prediction, or *all* possible alternatives must be considered.

8 Conclusions

In this paper, we have presented APIBENCHJ, a novel modular approach for automated benchmarking of large APIs written in Java. It benefits researchers and practitioners who need to evaluate the performance of API methods used in their applications, but also for evaluating the performance of APIs that they create. The presented approach is suitable for the Java Platform API, but also for third-party APIs accessible from Java.

This paper is a starting point into research on API benchmarking, as there exists no comparable approach for generic, large high-level object-oriented (Java) APIs. It addresses such quantitatively important issues as Just-In-Time compilation (JIT) and the influence of method parameters, assuming that appropriate method parameters are provided by existing heuristics [KOR09], manual specification, or recorded traces. This paper provided a first evaluation of APIBENCHJ on the basis of a subset of the Java Platform API including a comparison to the results of best-effort manual benchmarking.

Our next steps will include a large-scale study on the entire Java Platform API, and we also plan to automate the identification and quantification of parametric performance dependencies. In the future, APIBENCHJ can be extended by incorporating machine learning and other techniques of search-based software engineering for finding method parameters using APIBENCHJ 's plug-in mechanism, and for finding parametric dependencies [KKR09a]. It remains to be studied whether APIBENCHJ results can be used to *predict* the performance of a Java application in a real-life setting [KKR08], based on API usage by that application.

Acknowledgments: the authors thank Klaus Krogmann, Anne Martens and Jörg Henß and the entire SDQ research group for insightful discussions, suggestions and comments.

References

[Bel97] Doug Bell. Make Java fast: Optimize. *JavaWorld*, 2(4), 1997. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html, last visit: October 9th, 2009.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[BSW+00] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[Chi] Shigeru Chiba. Javassist (Java Programming Assistant). http://www.csg.is.titech.ac.jp/projects/index.html, last visit: October 9th, 2009.

[Cor08] Standard Performance Evaluation Corp. SPECjvm2008 Benchmarks, 2008. URL: http://www.spec.org/jvm2008/, last visit: October 9th, 2009.

[CP95] Cliff Click and Michael Paleczny. A Simple Graph-Based Intermediate Representation. In *ACM SIGPLAN Workshop on Intermediate Representations*. ACM Press, 1995.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[GP] William Griswold and Paul Phillips. UCSD Benchmarks for Java. http://cseweb.ucsd.edu/users/wgg/JavaProf, last visited October 9th, 2009.

[Int09] Intel Corporation. Intel VTune Performance Analyzer, 2009. http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-for-windows-documentation/, last visit: October 9th, 2009.

[KKR08] Michael Kuperberg, Klaus Krogmann, and Ralf Reussner. Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models. In *Proceedings of CBSE 2008*, volume 5282 of *LNCS*, pages 48–63. Springer-Verlag, Berlin, Germany, October 2008.

[KKR09a] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. *Transactions on Software Engineering*, 2009. accepted for publication, to appear.

[KKR09b] Michael Kuperberg, Martin Krogmann, and Ralf Reussner. TimerMeter: Quantifying Accuracy of Software Times for System Analysis. In *Proceedings of the 6th International Conference on Quantitative Evaluation of SysTems (QEST) 2009*, 2009. to appear.

[KOR09] Michael Kuperberg, Fouad Omri, and Ralf Reussner. Using Heuristics to Automate Parameter Generation for Benchmarking of Java Methods. In *Proceedings of the 6th International Workshop on Formal Engineering approaches to Software Components and Architectures, York, UK, 28th March 2009 (ETAPS 2009, 12th European Joint Conferences on Theory and Practice of Software)*, 2009.

[lin07] Linpack Benchmark (Java Version), 2007. URL: http://www.netlib.org/benchmark/linpackjava/, last visit: October 9th, 2009.

[Omr07] Fouad Omri. Design and Implementation of a fine-grained Benchmark for the Java API. Study thesis at chair 'Software Design and Quality' Prof. Reussner, February 2007.

[Pie] Darryl L. Pierce. J2ME Benchmark and Test Suite. http://sourceforge.net/projects/j2metest/, last visit: October 9th, 2009.

[sci07] Java SciMark 2.0, 2007. URL: http://math.nist.gov/scimark2/, last visit: Oct. 9th, 2009.

[Sta08] Standard Performance Evaluation Corp. SPECjAppServer2004 Benchmarks, 2008. http://www.spec.org/jAppServer2004/, last visit: October 9th, 2009.

[ZS00] Xiaolan Zhang and Margo Seltzer. HBench:Java: an application-specific benchmarking framework for Java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 62–70, New York, NY, USA, 2000. ACM Press.

Model-Driven Software Migration

Andreas Fuhr, Tassilo Horn
University of Koblenz-Landau
{afuhr,horn}@uni-koblenz.de

Andreas Winter
Carl von Ossietzky University Oldenburg
winter@informatik.uni-oldenburg.de

Abstract: In this paper we propose model-driven techniques to migrate legacy systems into Service-Oriented Architectures (SOA). The proposal explores how querying and transformation techniques on TGraphs enable the integration of legacy assets into a new SOA. The presented graph-based approach is applied to the identification and migration of services in an open source Java software system.

1 Introduction

Today, almost every company runs systems that have been implemented a long time ago and which are still adapted and maintained to meet current needs. Very often, adapting legacy software systems to new requirements also requires their migration to new technologies. Migrating legacy systems, i. e. transferring software systems to new environments *without* changing its functionality [SO08], enables already proven applications to stay on stream instead of passing away after some suspensive servicing [RB00].

A current technological advance promising better reusability of software assets is provided by *Service-Oriented Architectures (SOA)*. SOA is viewed as an abstract, business-driven approach decomposing software into loosely-coupled *services* enabling the reuse of existing software assets for rapidly changing business needs [GKMM04]. A service is viewed as an encapsulated, reusable and business-aligned asset coming with a well-defined *service specification* that provides an interface description of the functionality. The service specification is implemented by a *service component* which is realized by a *service provider*. Its functionality is used by *service consumers* [AGA+08].

Migrating legacy systems to services enables both, the reuse of already established and proven software components and the integration with newly created services, including their orchestration to support changing business needs. The application of model-driven techniques to software migration, presented here, is part of the SOAMIG project[1], which addresses the migration to Service-Oriented Architectures.

Software development and maintenance projects require a well-defined methodology. Major activities in software maintenance projects deal with legacy code. These include *legacy analysis*, i.e. understanding legacy systems and identifying reusable software assets and *legacy conversion*, i.e. migrating legacy assets. Current process models do not account for these activities, although approaches to software migration (e.g. [BS95]) are known.

This paper focuses on applying model-driven techniques to migrating legacy assets to SOAs. IBM's *SOMA method* [AGA+08] provides a process model for SOA development

[1] SOAMIG is funded by the Ministry of Education and Research (01IS09017C) cf. www.soamig.de .

and evolution, which serves here as a methodological framework to identify and extend migration activities and their technological support. Service-Oriented Modeling and Architecture (SOMA) includes seven incremental and iterative phases identifying, specifying and implementing services. In the first place, SOMA is designed to develop SOAs from scratch and does not provide support for integrating legacy assets.

The extension of SOMA towards migration, presented here, is based on model-driven strategies. Models represent different views on software systems including business process models, software architecture and programming code [WZ07]. Legacy analysis and conversion is based on queries and transformations on these models. In this paper, the *TGraph Approach* [ERW08] is applied as one possible model-driven technology. By migrating an exemplary service in the open source software GanttProject [Gan09], it will be shown how SOMA can be extended by model-driven technologies to provide a comprehensive methodology to SOA development, including a broad reuse of legacy code assets.

The paper is organized as follows: Section 2 describes the SOMA method and motivates where SOMA can be extended by model-driven techniques. Section 3 introduces the *TGraph Approach* as one possible technological space. In Section 4, the integrated method is applied to identify, specify, realize and implement one service by reusing GanttProject's legacy code. Section 5 briefly contrasts the integrated SOA migration approach presented here with current work in model-driven software analysis and migration. Finally, Section 6 summarizes and reflects the obtained results.

2 Service-Oriented Modeling and Architecture (SOMA)

IBM's SOMA method [AGA+08] is an iterative and incremental approach to design and implement service-oriented systems. It describes how to plan, design, implement and deploy SOA systems. SOMA is designed to be extensible in order to make use of additional, specialized techniques supporting project-specific needs. The following subsections shortly describe the seven SOMA phases and outline where they can be extended towards software migration.

Business Modeling: At the beginning of a project, the business is analyzed during this phase. Business goals and the business vision are identified, as well as business actors and business use cases.

SOA migration does not require to extend Business Modeling.

Solution Management: This phase adapts SOMA to the project needs. This includes choosing additional techniques to solve project-specific problems.

SOA migration requires to extend Solution Management: Customizing SOMA for migration requires the application of reengineering and migration techniques as depicted in the reminder.

Service Identification: In this phase, SOMA uses three complementary techniques to identify *service candidates*, i. e. functionality that forms a service.

Domain Decomposition is a top-down method decomposing the business domain into functional areas and analyzing the business processes to identify service candidates. *Goal-Service Modeling* identifies service candidates by exploring the business goals and sub-

goals. *Legacy Asset Analysis* finally explores the functionality of legacy systems. Documentation, APIs or interfaces are analyzed to identify service candidates. The source code is only analyzed coarse-grained. It is merely evaluated which functionality exists and not which components manifest the function. All three techniques are performed incrementally and iteratively. For each identified candidate, an initial service specification is created and a trace to the source of identification is established.

SOA migration requires to extend Service Identification: SOMA does not describe how to analyze legacy systems. In Section 4.3, we extend SOMA by reverse-engineering legacy code, which enables model-driven queries and transformations to identify service candidates including their code base.

Service Specification: This phase deals with describing the service design in detail. The initial service specification is refined, messages and message flows are designed and services are composed. This phase results in a comprehensive description of the service design. SOMA uses an UML profile for SOAs to describe the service design. Later, the service specification will be transformed into WSDL code in order to implement the service as a Web Service as is proposed by SOMA.

SOA migration requires to extend Service Specification: To gather the information needed for the service design, messages and message parameters can be derived from legacy code. We extend SOMA by queries to support design decisions in Section 4.4.

Service Realization: In this phase, it is decided which services will be implemented in the current iteration and it is constituted how to implement them.

After having chosen a set of services, the implementation strategy must be defined. Encapsulation of services allows to choose different ways to implement each service. Common strategies to form new service components include (1) implementation from scratch, (2) wrapping of legacy components or (3) transforming the required legacy components.

In SOMA, legacy functions are usually wrapped and then exposed as services. This has several drawbacks. The legacy system still requires maintenance, the wrapper needs to be created and requires maintenance during further evolution. Transforming the legacy code avoids this wrapping trap but requires appropriate transformation means. After deciding on transformation as implementation technique, legacy systems must be analyzed fine-grained. Functionality that is able to implement services has to be identified in the legacy code. In addition, it is important to clearly understand how this functionality is embedded in the legacy system, since it has to be separated to build a self-contained service.

SOA migration requires to extend Service Realization: SOMA does not consider how to implement services by reusing legacy code. In Section 4.5, a model-driven technique is presented to analyze legacy systems fine-grained in order to understand the implementation of legacy functionality.

Service Implementation: During Service Implementation, services are actually realized. According to the decisions derived in the Service Realization phase, services are developed, wrappers are written or legacy code is transformed. Finally, all services are orchestrated and message flows are established.

SOA migration requires to extend Service Implementation: SOMA does not include tech-

niques to transform legacy code into services. In Section 4.6 it is shown how transformations are used to transform legacy code into service implementations.

Service Deployment: The final phase of SOMA is Service Deployment. It deals with exposing services to the customer's environment. Final user-acceptance tests are executed and the SOA is monitored to verify that it performs as expected.

SOA migration does not require extensions of Service Deployment.

Concluding, five phases (solution management, identification, specification, realization and implementation) have been identified where SOMA must be extended to support migration of legacy systems into SOAs. The next section will describe which role model-driven development does play in extending SOMA. In Section 4, the extended SOMA method is exemplarily applied to the migration of an example Java application.

3 Model-Driven Development in Software Migration

Migrating legacy systems demands an integrated view on legacy code, software architecture and business processes [WZ07] to be able to identify and extract services. Therefore, an integrated representation of all these artifacts is essential. Model-driven approaches provide these technologies: (a) representation of models (metamodels), (b) querying models (query languages) and (c) transforming models (transformation languages).

Today, many model-driven approaches are known. Metamodels can be described by using the OMG's *Meta Object Facility* (MOF [OMG06]) or INRIA's KM3 [Ecl07]. Well-known transformation languages include QVT (Query/View/Transformation [OMG07]) or ATL (Atlas Transformation Language [ATL09]). All these approaches are suited for extending SOMA. However, in this paper, a graph-based approach is used which has already been applied in various reverse- and reengineering projects [ERW08].

The *TGraph Approach* is a seamless graph-based approach. Models are represented by graphs conforming to a *graph schema* (a metamodel). They can be queried with the graph query language *GReQL* (Graph Repository Querying Language [BE08]) and can be transformed using *GReTL* (Graph Repository Transformation Language [EH09]).

TGraphs are typed, attributed, directed and ordered graphs. Thus, they are based on a general graph model which allows appropriate tailoring for certain modeling purposes. In contrast to object-oriented modeling, edges are viewed as first-class objects. Hence, they can have attributes and traversal is always bidirectional. An API for accessing and manipulating TGraphs is given by the graph library *JGraLab*[2].

GReQL is a declarative graph query language and an enabling technology in reengineering, since various analyses of legacy systems can be mapped to graph queries [KW98]. One of GReQL's especially powerful features are *regular path expressions* which can be used to formulate queries that utilize the structure of interconnections between nodes and which support transitive closure.

GReTL is a Java framework for programming transformations on TGraphs making heavy use of GReQL to specify the mappings from source to target elements.

[2] http://jgralab.uni-koblenz.de

All these technologies are applied in Section 4 to identify, to extract and to migrate a service candidate of a Java example application within the SOMA methodology.

4 Extending SOMA by Model-Driven Techniques

The previous sections motivated the need of extending SOMA to enable the reuse of legacy software assets in software migration and shortly presented the TGraph approach. Our approach extends SOMA by applying model-driven techniques when appropriate.

Picking up the structure of introducing the SOMA phases in Section 2, the integrated approach is applied to the migration of one functionality of GanttProject into a Service-Oriented Architecture [Fuh09, HFW09]. GanttProject [Gan09] is a project planning tool. It manages project resources and displays project schedules as Gantt charts. GanttProject is implemented by about 1200 classes. The required migration is exemplified by identifying and migrating a service to *manage project resources* by transforming the appropriate legacy code.

4.1 Business Modeling

SOMA's first phase analyzes the current business situation. In this paper we focus on analysis and reuse of legacy software. Business modeling is not considered in detail, although it is important to analyze legacy business processes and to define processes to be supported by the SOA. Here, it is assumed that the business process of managing project resources shall be realized by the new SOA and its implementation will rely on GanttProject.

4.2 Solution Management

Solution Management adapts the SOMA method to the current project needs. Since GanttProject is a Java system, a TGraph representation for Java systems is required. Our Java 6 metamodel contains about 90 vertex and 160 edge types and covers the complete Java syntax. The GanttProject sources are parsed according to that metamodel, resulting in a graph of 274.959 nodes and 552.634 edges. This graph and the implicit knowledge on resource management, provide the foundation for service identification, service specification, service realization and service implementation.

4.3 Service Identification

The identification of services from legacy systems requires coarse-grained analysis. Firstly, the graphical user interface of GanttProject is explored and functionality to manage *project resources* is identified as one main feature of the software. Analyzing the GUI is only one entry point for identifying functionality and could be extended by additional explorations (e.g. test cases or documentation). Quickly scanning the legacy code then detects functionality providing the management of project resources.

Identifying pieces of functionality in legacy code is a challenging task and still an open research issue [KLS+07]. GReQL queries are used to identify functionality in the graph representation and corresponding GReTL transformations visualize the query result. String

search is used to detect possible code areas referring to “resources”. Further interconnections of code objects are specified by declarative path expressions. The resulting subgraph is transformed by GReTL into a TGraph conforming to a simple UML schema. Further XMI-based filters (cf. [EW06]) might be used to render these structures in UML tools.

```
1  VertexClass umlClass = createVertexClass("uml.Class",
2    "from t: V{Type} with t.name =~ \".*[Rr]esource.*\" reportSet t end");
3  createAttribute("name", umlClass, createStringDomain(),
4    "from t: keySet(img_uml$Class) reportMap t, t.name end");
5  createEdgeClass("uml.Association", umlClass, umlClass,
6    "from c: keySet(img_uml$Class), c2: keySet(img_uml$Class) "
7  + "with c <--{IsBlockOf} <--{IsMemberOf} <--{^IsBreakTargetOf,"
8  + "        ^IsContinueTargetOf,^IsTypeDefinitionOf,^IsClassBlockOf,"
9  + "        ^IsInterfaceBlockOf}* [<--{IsTypeDefinitionOf}] c2 "
10 + "reportSet c, c2 end",
11   "from t: $ reportMap t, first(t) end", "from t: $$ reportMap t, second(t) end");
12 createEdgeClass("uml.IsA", umlClass, umlClass,
13   "from c: keySet(img_uml$Class), c2: keySet(img_uml$Class) "
14 + "with c (<--{IsSuperClassOf} | <--{IsInterfaceOfClass}) "
15 + "        <--{IsTypeDefinitionOf} c2 "
16 + "reportSet c, c2 end",
17   "from t: $ reportMap t, first(t) end", "from t: $$ reportMap t, second(t) end");
```

Listing 1: GReTL transformation from Java to UML

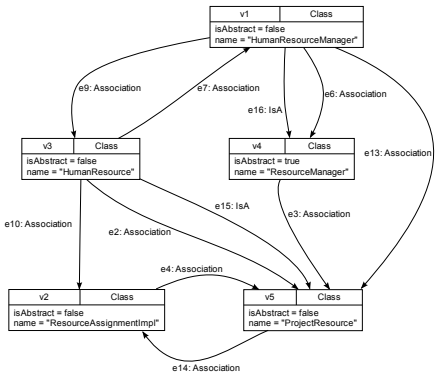
Listing 1 shows a GReTL transformation supporting coarse-grained legacy code analysis. For each legacy class or interface containing “resource” in the name, the transformation creates an UML-class node in the target TGraph (lines 1–4). In addition, associations are drawn between those class nodes whenever one node uses (e. g. by method calls or variable types) another node (5–11). Inheritance is visualized by “IsA” edges (12–17). These edges can be indirect relations since GReQL path expressions consider transitive closure. The transitive path expression in lines 7–9 (the part with $*$) matches indirect relations between classes. The visualized result of this GReTL transformation is shown in Figure 1a.

Looking at the result, the class HumanResourceManager implementing the interface ResourceManager is identified as functionality to manage project resources. Based on this information, an initial service specification for the service candidate IResourceManager is created and traces to the legacy code are noted (Figure 1b). Initial service operations for the new service are derived from the legacy interface. Only the *needed* functionality is transfered into the design. In this phase, no further information about the method signatures of the initial service specification is gathered. The following SOMA phases will specify the service in more detail.

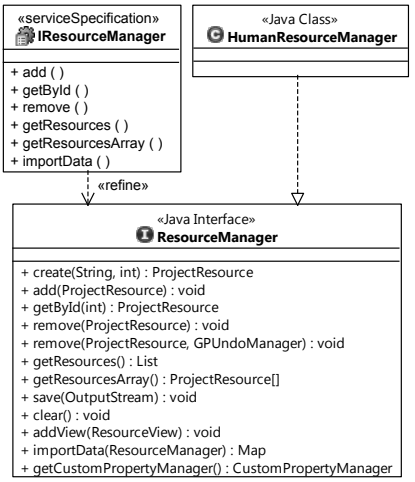
4.4 Service Specification

Service Specification refines the IResourceManager service specification. A *service provider* component is created which will later implement the service specification.

In addition, message flows are created to enable communication with the service. For *method parameters* in the legacy interface, *request messages* are created that are passed



(a) Visualization of classes and interfaces possibly providing functionality to manage resources



(b) IResourceManager service identified from legacy source code

Figure 1: Service Identification results

to the service. For *return types* in the legacy system, *response messages* are defined that will be returned by the new service. Request and response messages can be derived from legacy code.

Listing 2 shows a GReQL query taking an interface or class name as input and returning method parameters (lines 2–5) and return types (lines 6–9) as output. This information is used to derive message parameter types from legacy code.

```
1 let classname :="HumanResourceManager" in tup(
2   from hrmClass : V{ClassDefinition}, usedType : V{Type, BuiltInType}
3   with hrmClass.name = classname and hrmClass <--{IsClassBlockOf}<--{IsMemberOf}
     <--{IsParameterOfMethod} <--{IsTypeOfParameter} [<--{IsTypeDefOf}] usedType
4   reportSet (hasType(usedType, "BuiltInType")) ?
5     usedType.type : theElement(usedType<--&{Identifier}).name end,
6   from hrmClass : V{ClassDefinition}, usedType : V{Type, BuiltInType}
7   with hrmClass.name = classname and hrmClass <--{IsClassBlockOf} <--{IsMemberOf}
     <--{IsReturnTypeOf} [<--{IsTypeDefOf}] usedType
8   reportSet (hasType(usedType, "BuiltInType")) ?
9     usedType.type : theElement(usedType<--&{Identifier}).name end)
```

Listing 2: GReQL query retrieving method parameters and return types

The result of the specification phase is shown in the class diagram in Figure 2. The service specification now contains information about parameters (They are hidden in the ResourceManagerProvider component since they are already shown in the service specification). In addition, request and response messages are defined and one parameter type (HumanResourceType) for these messages is derived from legacy code.

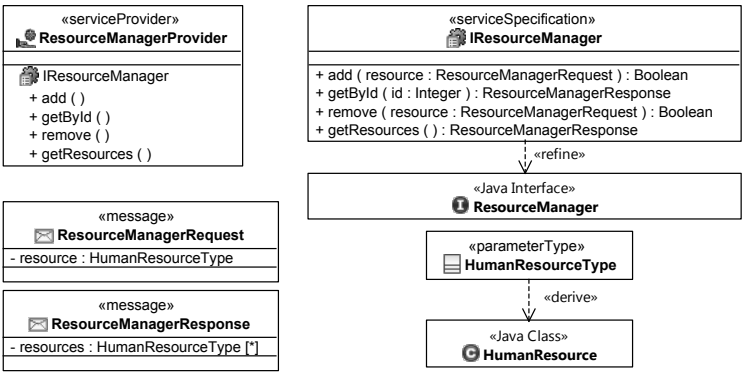


Figure 2: Detailed design of IResourceManager service

At the end of this phase, the design of the service itself is mostly completed. The next step will be to decide how the service should be implemented.

4.5 Service Realization

The first decision to make during Service Realization is how to implement the IResourceManager service. Model transformation approaches are also suited for code transformation. Thus, in this paper, the legacy code is transformed into a service implementation to provide the business functionality. If service realization by wrapping would have been decided on, wrappers could be generated analogously.

Service Identification has already identified one class in the legacy code that may provide functionality to the IResourceManager service: the class HumanResourceManager (short: HRM). The complete but minimal code realizing this functionality has to be determined and extracted, including all of the HRM dependencies. These include

- HRM calls methods of other classes (HRM $\rightarrow_{calls}$ method $\rightarrow_{isMemberOf}$ class),
- variables, parameters or return types (e.g. HRM $\rightarrow_{defines}$ variable $\rightarrow_{hasAsType}$ class),
- inheritance hierarchy (HRM $\rightarrow_{specializes}$ class or HRM $\rightarrow_{implements}$ interface).

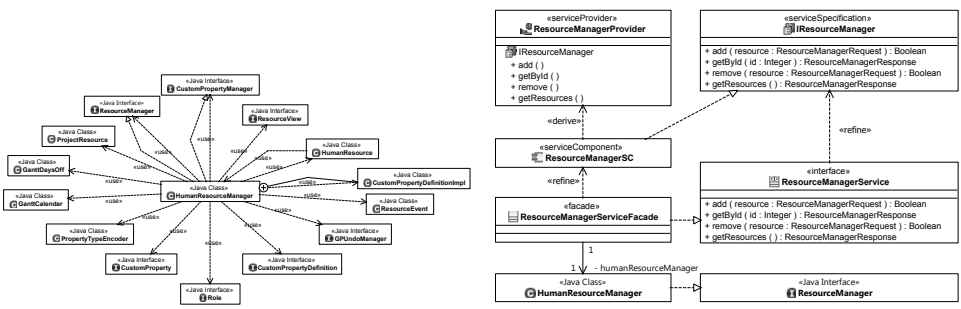
Listing 3 describes the GReQL query retrieving these dependencies. The core part of the query is the path expression in lines 4–9, defining how a dependency relation between caller and callee looks like in the Java metamodel. The query returns a list of all classes and interfaces that HRM depends on. In this example, only the directly related classes are retrieved. However, GReQL could retrieve the transitive closure of dependencies analogously. Figure 3a shows a (manually created) visualization of the query result.

Next, the business functionality must be integrated into the overall service design. This is done according to patterns proposed by Wahli [Wah07].

Figure 3b shows the application of these patterns to create a framework to integrate the legacy code which will be transformed in the next phase. The *service component* ResourceManagerSC implements the service specification. A facade pattern is used to im-

```
from hrmClass:V{ClassDefinition}, hrmMethod:V{MethodDefinition}, usedType:V{Type}
with hrmClass.name = "HumanResourceManager"
  and hrmClass <--{IsClassBlockOf} <--{IsMemberOf} hrmMethod
  and (hrmMethod ((<--{IsBodyOfMethod} <--{IsStatementOfBody} <--{^ReverseEdge}*
       <--{IsDeclarationOfInvokedMethod} -->{IsMemberOf} -->{IsClassBlockOf}
     ) | (<--{IsParameterOfMethod} <--{IsTypeOf}+ <--{IsTypeDefOf}
     ) | (<--{IsBodyOfMethod} <--{IsStatementOfBody} <--{^ReverseEdge}*
       <--{IsTypeOfVariable} <--{IsTypeDefOf}
     ) | (<--{IsReturnTypeOf} <--{IsTypeDefOf}) ) usedType
  or hrmClass ((<--{IsClassBlockOf}<--{IsMemberOf}<--{IsFieldCreationOf}
       <--{IsTypeOfVariable} <--{IsTypeDefOf}
     ) | ((<--{IsSuperClassOfClass} | <--{IsInterfaceOfClass}) <--{IsTypeDefOf}
     ) | ((<--{IsClassBlockOf} <--{IsMemberOf})+) ) usedType )
reportSet usedType end
```

Listing 3: GReQL query retrieving dependencies



(a) Dependencies of legacy HRM class

(b) Implementation design of IResourceManager

Figure 3: Service Realization: Designing the service implementation

plement the service component. The facade class will delegate service requests to the appropriate service implementation — in this example the HRM class and all its dependencies revealed by the GReQL query.

This phase finishes the design of the service. In the next step, this design will be implemented.

4.6 Service Implementation

Service Implementation realizes the IResourceManager service, e. g. as Web Service (as suggested by SOMA). Migrating identified source code (cf. Section 4.5) to realize the resource management service combines functionality provided by the IBM Rational Software Architect for WebSphere Software V7.5[3] (RSA) and TGraph technology.

Firstly, the code generation capabilities of the RSA are used to create WSDL code from the service specification which is later used to specify the service interface. Then, the design of the service framework (Figure 3b) is transformed into Java.

[3]IBM, Rational and WebSphere are trademarks of International Business Machines Corporation.

So far, the service lacks of business functionality, which is added by transforming legacy code into a service implementation. The GReQL query described in Listing 3 (Section 4.5) is used to mark the HRM class and all legacy software components it depends on.

The Java code generator of JGraLab is used to extract Java code for all marked classes of the TGraph. This results in Java classes implementing only the business functionality of the IResourceManager service. These classes are not connected to the service framework currently and so the facade class must be edited manually to delegate service requests to the HRM class. In addition, the facade class translates *message parameters* into *objects* known by the HRM class.

Finally, the *Web Service Wizard* of RSA is used to generate a fully functional Web Service. This wizard takes the WSDL interface description, the Java classes of the service framework and the service implementation and creates a Java EE Web Service.

4.7 Service Deployment

The Web Service created in the last subsection is deployed to the customer. This phase does not have to be extended by our approach. It concludes the migration.

5 Related Work

Whereas a plethora on publication on the development of Service-Oriented Architectures exists, migrating legacy systems to SOA is only addressed in a few papers. The SMART approach [Smi07] deals with the planning of SOA migration projects, but does not provide concrete migration or migration tool support. Correia et al. [CMHER07] and Fleurey et al. [FBB+07] describe general approaches of model-driven migration into a new technology not especially focused on SOA. Correia et al. describe a graph-based approach which mentions SOA as possible target architecture [Mat08]. In contrast to SOAMIG, this approach focuses on annotating functionality in legacy code instead of directly identifying services from source code. Marchetto and Ricca [MR08] propose an approach to migrate legacy systems into a SOA, step by step. However, this approach does not focus on model-driven techniques and uses wrapping as general migration strategy. Another approach focusing on wrapping is described in [Gim07].

In contrast to these approaches, the work presented here provides a coherent model-driven approach to software migration by integrating an established SOA forward-engineering approach with graph-based reengineering technologies. In addition, in SOAMIG software systems are viewed at all levels of abstraction including business processes and code.

6 Conclusion and Future Work

In this paper, we described a model-driven approach to migrate legacy systems, extending IBM's SOMA method. The approach was applied to the migration of a functionality of GanttProject towards a Service-Oriented Architecture. This example demonstrated the identification and specification of services by analyzing legacy code, the identification of responsible functionality in legacy code and the transformation of legacy code into a service implementation. As result, a fully functional Web Service was generated whose

business functionality was implemented by transforming legacy code.

The example presented in this paper should be understood as first technical proof-of-concept and not as fully developed method. The approach must be extended and the techniques must be improved. E.g. one issue is the replacement of the string-based service identification technique (which fails when source code does not follow some naming conventions) by a dynamic approach. As part of the SOAMIG project, we are currently exploring the possibility of simulating the execution of business processes while tracing source code execution. Based on these traces, source code might be mapped to processes, supporting service identification.

Another issue is the application of the approach on systems that are not written in Java. In addition to plain architectural migrations as presented in this example, languages (e.g. COBOL $\rightarrow$ Java) must be migrated in language migration projects, too. All TGraph techniques explained in this paper are generic and will work for other languages if metamodels are provided. Currently, a metamodel for COBOL is being developed and will enable our approach to cope with legacy COBOL systems as soon as it is finished. Research will also address code transformation based on these metamodels.

In contrast to "transformation capabilities" of modern tools like IBM's Rational Software Architect or Borland's Together Architect, the TGraph approach offers an integrated view on all models and allows to process all needed queries on one repository. This will allow us to create one homogeneous workflow instead of handling different types of results from different sources leading to compatibility issues.

The results of this ongoing research will have to be confirmed on larger examples in future. In collaboration with our industrial project partners, their Java and COBOL systems will be migrated into Service-Oriented Architectures.

Summarizing, the presented approach already showed first interesting results and promises to improve model-driven migration in future.

Acknowledgements

We want to express our thanks to Rainer Gimnich, IBM Software Group, Frankfurt/Main, Germany for his support in understanding SOMA and various fruitful discussions on applying SOMA in software migration.

References

[AGA+08] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. SOMA: A Method for Developing Service-Oriented Solutions. *IBM Systems Journal*, 47(3):377–396, 2008.

[ATL09] ATLAS Group. ATL User Guide, 2009.

[BE08] D. Bildhauer and J. Ebert. Querying Software Abstraction Graphs. In *Working Session on Query Technologies and Applications for Program Comprehension*, 2008.

[BS95] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems, Gateways, Interfaces & The Incremental Approach.* Morgan Kaufmann, 1995.

[CMHER07] R. Correia, C. Matos, R. Heckel, and M. El-Ramly. Architecture Migration Driven by Code Categorization. In *Software Architecture, First European Conference, Spain:*

Proceedings, volume 4758 of *Lecture Notes in Computer Science*. Springer, 2007.

[Ecl07] Eclipse. KM3. `http://wiki.eclipse.org/KM3`, 2007.

[EH09] J. Ebert and T. Horn. The GReTL Transformation Language. Projectreport, Koblenz, 2009.

[ERW08] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering: The TGraph Approach. In R. Gimnich, U. Kaiser, J. Quante, and A. Winter, editors, *10th Workshop Software Reengineering*, volume 126 of *GI-Edition Proceedings*, pages 67–81, Bonn, 2008. Ges. f. Informatik.

[EW06] J. Ebert and A. Winter. Using Metamodels in Service Interoperability. In *Postproceedings of 13th Annual International Workshop on Software Technology and Engineering Practice (STEP'05)*, pages 147–156, 2006.

[FBB+07] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J. M. Jezequel. Model-driven Engineering for Software Migration in a Large Industrial Context. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *International Conference on Model Driven Engineering Languages and Systems*, pages 482–497, Berlin, 2007. Springer.

[Fuh09] A. Fuhr. Model-driven Software Migration into a Service-oriented Architecture. Bachelorthesis, Mainz, 2009.

[Gan09] GanttProject. The GanttProject. `http://ganttproject.biz/`, 2009.

[Gim07] R. Gimnich. SOA Migration: Approaches and Experience. *Softwaretechnik-Trends*, 27(1), 2007.

[GKMM04] N. Gold, C. Knight, A. Mohan, and M. Munro. Understanding Service-Oriented Software. *IEEE Software*, 21(2):71–77, 2004.

[HFW09] T. Horn, A. Fuhr, and A. Winter. Towards Applying Model-Transformations and -Queries for SOA-Migration. In *Workshop MDD, SOA und IT-Management*, 2009.

[KLS+07] K. Kontogiannis, G. A. Lewis, D. B. Smith, M. Litoiu, H. Müller, S. Schuster, and E. Stroulia. The Landscape of Service-Oriented Systems: A Research Perspective. In *Proceedings of the International Workshop on Systems Development in SOA Environments*. IEEE Computer Society, 2007.

[KW98] B. Kullbach and A. Winter. Querying as an Enabling Technology in Software Reengineering. In *Proceedings of the 3nd European Conference on Software Maintenance and Reengineering*, pages 42–50. IEEE Computer Society, Los Alamitos, 1998.

[Mat08] C. Matos. Service Extraction from Legacy Systems. In D. Hutchison, H. Ehrig, R. Heckel, T. Kanade, and J. Kittler, editors, *Graph Transformations*, volume 5214, pages 505–507, Berlin, Heidelberg, 2008. Springer-Verlag.

[MR08] A. Marchetto and F. Ricca. Transforming a Java Application in a Equivalent Web-Services Based Application: Toward a Tool Supported Stepwise Approach. In *Proceedings Tenth IEEE International Symposium on Web Site Evolution, Beijing, China (WSE)*. IEEE Computer Society, 2008.

[OMG06] OMG. Meta Object Facility (MOF) 2.0: Core Specification – formal/06-01-01, 2006.

[OMG07] OMG. Meta Object Facility (MOF) 2.0: Query/View/Transformation Specification – Final Adopted Specification ptc/07-07-07, 2007.

[RB00] V. T. Rajlich and K. H. Bennett. A Staged Model for the Software Life Cycle. *Computer*, 33(7):66–71, 2000.

[Smi07] D. B. Smith. Migration of Legacy Assets to Service-Oriented Architecture Environments. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 174–175. IEEE Computer Society, 2007.

[SO08] H. M. Sneed and S. Opferkuch. Training and Certifying Software Maintainers. In *12th European Conference on Software Maintenance and Reengineering (CSMR), Athens, Greece*, pages 113–122. IEEE Computer Society, 2008.

[Wah07] U. Wahli. *Building SOA Solutions Using the Rational SDP*. IBM Redbooks. 2007.

[WZ07] A. Winter and J. Ziemann. Model-based Migration to Service-oriented Architectures: A Project Outline. In H. M. Sneed, editor, *CSMR 2007, 11th European Conference on Software Maintenance and Reengineering, Workshops*, pages 107–110, 2007.

Towards an Architectural Style for Multi-tenant Software Applications

Heiko Koziolek

ABB Corporate Research
Industrial Software Systems
Wallstadter Str. 59, 68526 Ladenburg, Germany
heiko.koziolek@de.abb.com

Abstract: Multi-tenant software applications serve different organizations from a single instance and help to save development, maintenance, and administration costs. The architectural concepts of these applications and their relation to emerging platform-as-a-service (PaaS) environments are still not well understood, so that it is hard for many developers to design and implement such an application. Existing attempts at a structured documentation of the underlying concepts are either technology-specific or restricted to certain details. We propose documenting the concepts as a new architectural style. This paper initially describes the architectural properties, elements, views, and constraints of this style. We illustrate how the architectural elements are implemented in current PaaS environments, such as Force.com, Windows Azure, and Google App Engine.

1 Introduction

A multi-tenant software application is a special type of hosted software that individually serves different tenants (i.e., organisations, such as companies or non-profit groups) from a single instance [CC06a]. Here, a single instance means that the software runs on the same infrastructure, is implemented from the same code base, and can be updated centrally. Currently, several platform-as-a-service (PaaS) environments, such as Force.com, Windows Azure, and Google App Engine, are emerging [AFG+09]. They are able to host multi-tenant software in large data centers and shall save software vendors costs for setting up and maintaining a hardware/software infrastructure.

Following the success of Salesforce [WB09], multi-tenant architectures have been identified as a potentially critical competitive advantage over classical single-tenant architectures in certain domains [CC06a]. However, in spite of the appearing platforms, it is still hard for software developers to design and implement efficient multi-tenant architectures [WGG+08]. Due to the missing documentation and formalisation of the underlying architectural concepts, many developers do not possess a clear view on the necessary design decisions and architectural trade-offs of a multi-tenant application.

Existing attempts on providing a structured documentation of the architectural concept

underlying a multi-tenant software architectures either depend on specific technologies (e.g., [CC06b]) or focus on restricted aspects, such as the database layer (e.g., [AGJ+08]). Descriptions of existing multi-tenant software architectures (e.g., [WB09]) help to achieve an initial comprehension but are difficult to transfer to other applications. A more abstract and technology-agnostic perspective is needed to understand the involved design decisions and architectural trade-offs.

We propose documenting the concepts and constraints underlying a multi-tenant software architecture as a new *architectural style* (the so-called SPOSAD style: **S**hared, **Po**lymorphic, **S**calable **A**pplication and **D**ata). Classical styles, such as client/server or pipe-and-filter, have been documented decades ago and are still being used to structure new software applications [TMD09]. Recent architectural styles, such as REST (REpresentational State Transfer) for the WWW [FT02] and SPIAR (Single Page Internet Application aRchitectural style) for AJAX application [MvD08] help clarifying the architectural features of web applications. In this paper, we propose documenting the features of multi-tenant software architectures as an extension of the n-tier architectural style. The described concepts shall be reusable and guide the development of new multi-tenant software applications.

The contribution of this paper is an initial description of the architectural elements and properties of a multi-tenant software architecture. We set the architectural concepts in context to the capabilities of current PaaS environments. We also provide an initial description of the design decisions to be made in the application tier and in the database tier. Ultimately, our description shall be evolved to a formal documentation of a new architectural style for multi-tenant software applications.

This paper is organized as follows: Section 2 describes three commercial PaaS environments, which are built according to multi-tenant software architectures. Section 3 then briefly recalls concepts about architectural styles that are relevant in the context of this paper. Section 4 provides an initial description of the SPOSAD style and lists architectural properties, elements, views, and constraints. Section 5 discusses the style, before Section 6 reviews related work, and Section 7 concludes the paper.

2 Multi-tenant Architectures in PaaS Environments

Several PaaS environments are currently being developed [AFG+09]. These environments are built on top of large data centers and shall help software developers to develop cloud-based applications. In the following, we will briefly summarize the most important architectural elements of three PaaS environments, before we derive an architectural style from their commonalities in later sections.

Force.com: With the force.com platform developers may build applications on top of the salesforce.com infrastructure. On a high abstraction level, the platform is built according to an n-tier architecture [WB09] comprising a presentation tier (using web browsers), an application tier, and a data tier.

Clients access the application tier of force.com according to the REST style. Each tenant is served by application instances originating from the same code base. Salesforce manages

updates of this code base centrally. Tenants can customize the application user interface (forms), business logic (workflows), and data (customized tables) by specifying meta-data stored in the so-called Universal Data Dictionary (UDD). A runtime engine generates tenant-specific application code from this meta-data. Thus, the application is considered 'polymorphic', as it appears and behaves differently for the clients of each tenant.

Through the application tier, all tenants access the same logical database in the data tier. All tenant data is stored in a single table, which can be partitioned among multiple machines. Besides a tenant id column, the table contains 500 customizable columns (varchar datatype) for storing arbitrary data (i.e., a universal table layout [AGJ+08]). Information about tenant-specific entities is stored in an additional 'objects' meta-data table, while information about tenant-specific fields is stored in an additional 'fields' meta-data table.

Windows Azure: The Windows Azure platform by Microsoft allows deploying and running ASP.NET and WCF application in Microsoft data centers [Cha09]. The data centers run the Windows Azure Hypervisor and modified versions of Windows Server 2008 on a large number of virtual machines. The platform follows a three-tier structure.

Clients, such as browsers or web services, access the application tier using REST or SOAP. In the application tier, applications with a UI are implemented as so-called 'web-roles', while background application are implemented as so-called 'worker-roles'. Web-roles and worker-roles may interact asynchronously using queues. They are ideally stateless and may be run in a configurable number of instances. Load balancers can distribute requests among those instances. Tenants can implement UI customizations using MS Silverlight and business logic customizations using Windows Workflows as demonstrated in [Cum09].

Web/Worker-roles can either access the non-relational, horizontal scalable Windows Azure storage or slightly customized versions of the MS SQL server (SQL Azure). The Windows Azure storage features blobs, non-relational tables, and queues for data persistency. Blobs are binary large objects up to 50 GB large, while the tables can hold a hierarchical structure of key/values pairs. Queues handle interaction between web- and worker roles by storing data portions. Tenant-specific data can either be stored in the Windows Azure storage using tenant IDs for delimination or in SQL Azure with each tenant accessing its own database instance.

Google App Engine: Google's App Engine (GAE) [Goo09] enables developers to run Java or Python applications in Google's data centers. Several web frameworks, such as Django, Cherrypy, or pylons run on GAE and assist developers in implementing their applications.

Clients access the application tier of GAE using REST. Besides responding to web requests, GAE also allows to run so-called 'scheduled tasks' as possibly periodic background tasks. Web applications and background tasks might interact asynchronously using queues. Developers using GAE do not gain control on the VMs running their software, which shall relieve them from the administration tasks. GAE features built-in auto-scaling, load balancing, and fail-over mechanisms between identical implementation instances in the application tier.

Applications may store data in a non-relational structure called Google Big Table, which shall be able to handle large-scale applications and store petabytes of data. The GAE

storage features a proprietary query engine with the Google Query Language that allows transactions. The Big Table does not have a schema, the structure of the contained data entities must be provided and enforced by the application code.

3 Architectural Styles

To describe the architectural concepts in this paper, we use the terminology of Perry and Wolf [PW92]. They define an architecture as a configuration of architectural elements - processing (i.e., components), connectors, and data - constrained in their relationships in order to achieve a desired set of architectural properties.

According to Fielding [FT02], an architectural style is a coordinated set of architectural constraints that restricts the roles of the architectural elements and the allowed relationships among those elements within any architecture that conforms to that style. Basic architectural styles are for example client/server, n-tier, pipe-and-filter, and code on demand. More complex styles, which build on and extend the basic styles are model-view-controller [KP88] for GUIs, REST for the WWW [FT02], and SPIAR for AJAX applications [MvD08].

REST and SPIAR are architectural styles related to multi-tenant architectures. REST induces a constrained client/server architecture with focus on the communicated data elements. The style prescribes the use of resources (i.e., the target of hyperlinks), resource identifiers (e.g., URLs), representations (e.g., HTML documents, JPEG images), and meta-data. Two constraints for the architecture are a synchronous request/response communication between client and server as well as stateless and context-free interaction for scalability.

SPIAR targets client / server architectures with rich user interfaces and was deduced from AJAX applications. The style constraints the architecture by prescribing asynchronous interaction between client and server, delta-communication (i.e., only state-changes are transferred to reduce network traffic), and component-based user interface for more interactivity.

Both the REST and SPIAR style (if stateless) might be used in a multi-tenant architecture. However, they are not sufficient to describe such architectures. Multi-tenancy puts additional constraints on the code to be used at the application tier and the data elements held in the data tier as described in the next section.

4 The SPOSAD Style

In this section, we first describe the essential architectural properties of multi-tenant applications (Section 4.1). Then, Section 4.2 focusses on the architectural elements of the SPOSAD style, before Section 4.3 illustrate the style with architectural views. Finally, Section 4.4 lists the architectural constraints induced by the SPOSAD style.

4.1 Architectural Properties

First, we discuss the architectural properties that relate to the goals of multi-tenant software. They mainly focus on extra-functional properties to be achieved by the style. These properties can also be viewed as requirements for a multi-tenant architecture.

Resource Sharing: the main motivation for an multi-tenant architecture is to save development and administration costs for hosted software by serving multiple tenants from the same code base and shared data repositories. The term 'resource' is used here in a broader sense. The software instance shall share hardware and software resources, such as hardware infrastructure, virtual machines, operating systems, databases, and code. On the other hand the software shall share development and maintainance resources by using a single code base for multiple tenants.

Scalability: because multiple tenants with potentially thousands of clients shall be served, scalability is an important architectural property for a multi-tenant architecture. Scalability refers to the ability of a system to either handle growing amounts of work in a graceful manner or to be readily enlarged. For example, this means that the response time of an application remains stable if the workload (i.e., the number of users concurrently using the system) is increased. Due to the inherent limits of scaling up (i.e., adding resources to a single node), the ability to seamlessly scale out (i.e., adding more nodes) is a typical architectural property of a multi-tenant system.

Maintainability: while many hosted, single-tenant application rely on different code bases for tenant customizations, a multi-tenant architecture relies on a shared code basis for several tenants. This feature helps to decrease the maintenance effort for the software, because bugs only need to be fixed once and updates can be installed centrally. The multi-tenant design with a shared database also reduces costs for database administration and maintenance, which does not have to be executed for each tenant.

Customizability: the ability to incorporate tenant-specific customizations is another important property of a multi-tenant architecture. Because of the shared application code base and shared database it is not trivial to allow tenants to adapt the application's business logic and data to the requirements of their clients. A well-designed multi-tenant architecture is able to find a good trade-off between resource sharing and user customizability.

Usability: besides changing the business logic and the data of an application, also the user interface shall be configurable through tenant-specific customizations. It allows different tenants to create their own branding for an application.

4.2 Architectural Elements

The architectural elements described in the following are structured according to the categories by Perry and Wolf [PW92], processing elements, connecting elements, and data elements. A processing view of the style is depicted in Fig. 1 and will be explained in Section 4.3.

4.2.1 Processing Elements (Components)

Components process the data elements of the system and communicate via connectors. The following components are most relevant for the SPOSAD style.

On the client tier, users interact with the system using a client application, which can be a *web browser* for displaying HTML documents and images or a *rich client* for more sophisticated user interfaces. The client application accepts user inputs and handles the whole user interaction. It is not different from client applications in typical n-tier architectures.

On the application tier, multiple identical *application threads* execute the business logic of the application. Multiple threads or processes allow the application to scale out. It is possible to distribute the threads or processes to multiple physical processing nodes and to distribute the user requests among them. The threads can be considered polymorphic, because they may appear and behave differently for different tenants based on the meta-data they access. However, the code of the application threads comes from a single code base, which bears no tenant-specific extensions.

A *meta-data manager* handles the customization of the application threads with tenant-specific meta-data. This data may for example relate to tenant-specific input forms, UI brandings, business logic, workflows, and access privileges. There are different possibilities to implement such an application customization mechanism. For example, the meta-data manager might generate tenant-specific application code from a common code base on-the-fly. Or the meta-data manager might simply be responsible for retrieving tenant-specific meta-data and the application adapts itself to this data.

To ensure horizontal scalability, multiple application threads are running concurrently. A *load balancer* distributes client requests to these threads. Many load balancing strategies with different benefits and drawbacks are known. As the application threads shall be stateless, the load balancer can distribute subsequent requests by the same client to different application threads. The load balancer can also be responsible for auto-scaling, i.e. starting new threads upon increasing workload and stopping running threads upon decreasing workload (also known as elasticity).

On the application tier, the components processing the user requests are arranged according to a pipe-and-filter style. Therefore, additional components, such as *caches* or *proxies*, might process the user requests. However, these components can be considered optional in the SPOSAD style.

The database tier contains a *multi-tenant data resource* and a *meta-data resource*. For maximal resource sharing, a single database application should serve all tenants to save processing overheads, memory footprint, and administration costs. Different options for the data and schema layout for multi-tenant applications are discussed in the section about data elements. Hosting a large amount of tenants in the same database results in the need to partition the database and to store the data onto multiple physical nodes.

In addition to the components described so far, other optional components might be present in a multi-tenant architecture. For example, multiple tenants on the same resource require measures for user authentication and authorization and security measures (e.g., encryption). If the application shall be licensed according to a pay-per-use scheme, components

for metering application usage and billing users are required.

4.2.2 Connecting Elements (Connectors)

All components in the SPOSAD style are connected by *procedure calls*. The communication can for example follow the REST style with synchronous calls. Clients requests service from the server, the application threads carry out the service based on the data in the database and send responses back to the clients. Following the SPIAR style that is used in AJAX applications, the communication might also be asynchronous with the client requesting additional service upon state changes.

The connections might involve additional resolvers (e.g., DNS lookup) or tunnels (SOCKS, SSL after HTTP) as additional, optional connectors. Because the component topology follows the pipe-and-filter style, the communication can be flexibly extended.

The communication between the application tier and the database tier might be synchronous (e.g., for simple, short queries) or asynchronous (for long running queries). Asynchronous communication might result in more efficient resource utilization as the application threads do not have to wait for the database to respond and already serve further user requests.

4.2.3 Data Elements

The data elements in the system are the messages exchanged by the components as well as the data stored in the database tier. The *messages* sent by the components might be RESTful. Each client request at least has to include a tenant id, so that the client only sees tenant-specific data and gets a tenant-specific application.

The data stored in the database at least consists of *tenant-specific application data* and *meta-data*. When designing the data storage, the goal of the SPOSAD style is to share an adequate amount of resources.

Chong et al. [CC06b] discuss the benefits and drawbacks of different data architectures. Using a separate database per tenant is easy to implement and beneficial for security purposes, but it there is limited resource sharing and thus high costs for hardware, database administration, and backup procedures. Using a shared database, but per-tenant schemas reduces hardware costs and maintenance effort, but raises security issues and complicates backup procedures. Even more efficient is using a single schema for all tenants. Such an approach has the lowest memory overhead and low administration costs, but requires special measures for arranging the data and ensuring security. In general, the more tenants need to be served by the multi-tenant application, the better is a shared schema approach, because it reduces the memory and maintenance overheads and allows exploiting economics of scale.

Aulbach et al. [AGJ+08] have compared different schema-mapping techniques for multi-tenant databases. A *private table layout* provides a single table per tenant. In the presence of many tenants, this can induce a memory overhead for storing the individual table structures. An *extension table layout* provides a single table for common tenant data, and an additional table per tenant for schema extensions. If many tenants customize the schema,

this again results in a high number of required tables and additionally requires joins when accessing the data.

An *universal table layout* provides a single table for storing data by all tenants. This layout is for example used by the force.com architecture. It contains multiple columns without fixed datatypes for storing tenant-specific data. This layout might be more efficient, because there is no need for expensive joins to reconstruct the logical schema. However, it requires storing many null values and may require type convertions. In a *pivot table layout* each table contains tenant and row ids and a single column for a specific data type. While this layout reduces the need to store null values, it again requires joins to reconstruct the logical schema.

The SPOSAD style requires the architect to use a data architecture where resources are shared. For high scalability, the style also requires the architect to use a data partitioning scheme to physical servers that best allows for scaling out large amounts of data. This can for example involve tenant-specific partitions or local partitions for different user groups of a single tenant.

4.3 Architectural Views

Different views can illustrate the interplay of the architectural elements described in Section 4.2. Fig. 1 depicts a processing view of an architecture implemented according to the SPOSAD style. The figure shows the noticeable relation of the SPOSAD style to the n-tier architectural style, as it features a client, application, and database tier.

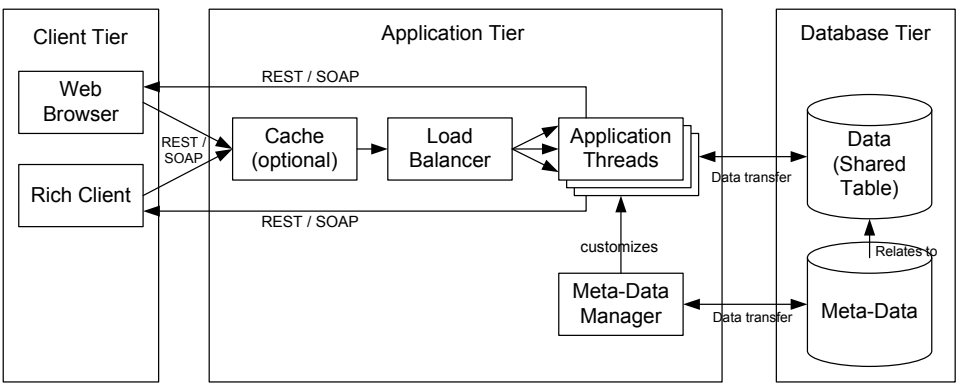


Figure 1: Processing View of the SPOSAD Style

Clients using web browsers or rich client applications interact with the application tier, which in turn accesses the database tier. The polymorphic application threads are the heart of the application tier. A load balancer directs user requests to them. The meta-data manager ensures that tenant-specific customizations are included in the application. The data tier differs from traditional n-tier architecture in the arrangement of the data, which is

stored in a multi-tenant database that allows maximal resource sharing.

The figure neglects many details of an architecture implemented according to the SPOSAD style and focuses on the elements that differentiate multi-tenant architecture from n-tier architectures. Additional components for authentication, authorization, connection pooling, security handling etc. are required. Such an application typically runs on application servers and may be hosted on virtual machines. The physical topology, i.e., the allocation of the components to hardware nodes, is also not depicted here.

4.4 Architectural Constraints

The SPOSAD style induces the following architectural constraints, which restrict architects when designing such a system.

Single code base: The application is developed in a single code base. Tenant-specific extension shall be made only via meta-data, but not by changing the code. This constraint complicates implementing the application, because mechanisms for meta-data driven tenant customizations have to be found. On the other hand this constraint enables sharing of development efforts and allows for central bug fixes and updates.

Shared resources in the database tier: Architects may not use isolated, tenant specific data layouts in the database tier as in typical n-tier applications. The style mandates sharing resources to reduce costs for database administration and backup procedures as well as hardware.

Customizable application: The application threads must allow for tenant-specific extensions using meta-data. While resource sharing is the most desirable property of the SPOSAD style, tenant-specific customizations are a necessity from a business perspective, as tenants will not accept standard solutions in many cases.

Stateless application tier: Client specific state, such as transactional data or inputs of users forms, may not be stored in the application tier. The application threads shall be stateless to allow serving the same client with different threads in subsequent requests. This constraint allows for efficient usage of the processing resources, as the application threads do not have to wait for user inputs of a specific client, but can process requests by other users in the mean time. Client specific state thus has to be stored at the client-side (e.g., using cookies) or in the database tier.

5 Discussion

Architects have to make several decisions and trade-offs when developing multi-tenany applications. They have to define the *degree of customization* that the application should support. More customisability implies more complicated development and makes the use of shared resources more difficult. Thus, highly customizable applications are not well suited for a multi-tenant architecture.

Architects have to work out concepts to deal with *security issues*. Hosting business-critical data of multiple tenants in the same infrastructure or even the same database table requires special measures for keeping the data logically isolated (e.g., using encryption).

When multiple tenants are using the same infrastructure, it has to be ensured that the application threads of one tenant do not interfere with application threads by other tenants (e.g., by crashing the underlying VM or decreasing performance). *Reliability measures* might include application thread replication and the isolation of performance-intensive application tasks onto individual VMs. For example, Windows Azure replicates each web and worker role (i.e., application threads) three times.

The PaaS environments described in Section 2 are already built according to the SPOSAD style or at least support building multi-tenant application and therefore should be considered by architects when making *build or buy decisions* for a multi-tenant infrastructure. Force.com includes a meta-data manager that generates the application thread code during runtime from meta-data and manages all tenant data according to a universal table layout. Azure uses web and worker roles as application threads and features a horizontally scalable storage solution with the Windows Azure tables. GAE runs application threads implemented in Java or Python code, but does not support meta-data management out-of-the-box. Like in Azure, data storage is handled in a horizontally scalable, non-relational table structure.

It is furthermore helpful to delimit multi-tenant architectures from single-tenant, n-tier architectures to make their special features better comprehensible. For example, an application such as Hotmail hosts the data of multiple tenants in the same infrastructure, but does not allow for tenant-specific customizations using meta-data. The software as a service solution by SAP for small companies called BusinessByDesign hosts the clients of each tenant on a dedicated physical machine. Thus, it can be considered a single-tenant solution.

6 Related Work

Due to the novelty of PaaS environments and cloud platforms, there is only limited scientific research for multi-tenant architectures.

Chong and Carraro from Microsoft discuss the business rationale of SaaS applications and describe their high level architectural concepts [CC06a]. Level 4 of their SaaS maturity model (i.e., a scalable, configurable, multi-tenant efficient application) can be considered conforming to the SPOSAD style sketched in this paper. The same authors have also discussed the benefits and drawbacks of different database layouts for multi-tenancy applications [CC06b]. However, they do not describe a reusable architectural style.

Weissman [WB09] provides an overview of the force.com architecture, which realises many concepts for multi-tenancy. His description is tied to a specific platform and thus not easily transferable to other multi-tenant architectures.

Aulbach et al. [AGJ+08] provide a database centric view on multi-tenant architectures. They evaluate the performance properties of different flexible schemas for multi-tenant

applications and propose a new, more efficient schema. Their analysis lacks an evaluation of the scalable storage solutions of current PaaS environments. Furthermore, they completely neglect the application layer of multi-tenant applications.

Wang et al. [WGG+08] proposed a framework for implementing multi-tenant applications. They describe patterns for security, performance, and administrations isolation in such architectures and sketch customization concepts. Furthermore, they identify performance bottlenecks and optimization approaches for such applications. However, they neglect the application tier in their investigation.

Kwok et al. [KM08] deal with capacity planning in multi-tenant applications and propose a method for determining the optimal allocation of application threads to physical nodes. Mietzner et al. [MLP08] extend the service component architecture (SCA) to be able to describe multi-tenant applications.

In the area of architectural styles, Perry and Wolf [PW92] laid the foundation for describing reusable styles. Taylor et al. [TMD09] provide an up-to-date description of the most important documented architectural styles. Among them are REST [FT02] for the WWW and SPIAR [MvD08] for AJAX applications.

7 Conclusions

This paper has described the component, connectors, and data elements of a typical multi-tenant software architecture and discusses various properties and constraints of such an architecture. The description shall ultimately lead to the description of a new architectural style for multi-tenancy applications. The paper has also put the described architectural elements in context of three current PaaS environments.

The identification of a new architectural style helps developer in creating future multi-tenant software applications. While the emerging PaaS environment are well-suited for implementing such applications, there are still many design decisions at the application tier and the database tier that have to be made for each application. An architectural style can help developers in understanding the architectural trade-offs and the implications of their decisions.

As future work, we plan to formalize the style description further. We will provide different views of the style and describe further architectural constraints. Furthermore, we will analyse more existing multi-tenant applications for their implementation of the SPOSAD style concepts.

References

[AFG+09] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report 2009-28, UC Berkeley, 2009.

[AGJ+08] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-

tenant databases for software as a service: schema-mapping techniques. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'08)*, pages 1195–1206, New York, NY, USA, 2008. ACM.

[CC06a] Frederick Chong and Gianpaolo Carraro. Architecture Strategies for Catching the Long Tail. Technical report, Microsoft Corporation, http://msdn.microsoft.com/en-us/library/aa479069.aspx, April 2006. Last visited 2009-10-09.

[CC06b] Frederick Chong and Gianpaolo Carraro. Multi-tenant Data Architecture. Technical report, Microsoft Cooperation, http://msdn.microsoft.com/en-us/library/aa479086.aspx, June 2006. Last visited 2009-10-09.

[Cha09] David Chappell. Introducing the Windows Azure Platform. Technical report, DavidChappell & Associates, http://go.microsoft.com/fwlink/?LinkId=158011, August 2009.

[Cum09] Cumulux. Project Riviera Website. http://code.msdn.microsoft.com/riviera, September 2009. Last visited 2009-10-09.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[Goo09] Google. App Engine. http://appengine.google.com, October 2009. Last visited 2009-10-09.

[KM08] Thomas Kwok and Ajay Mohindra. Resource Calculations with Constraints, and Placement of Tenants and Instances for Multi-tenant SaaS Applications. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 633–648, Berlin, Heidelberg, 2008. Springer-Verlag.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[MLP08] Ralph Mietzner, Frank Leymann, and Mike P. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 156–161, Washington, DC, USA, 2008. IEEE Computer Society.

[MvD08] Ali Mesbah and Arie van Deursen. A component- and push-based architectural style for AJAX applications. *J. Syst. Softw.*, 81(12):2194–2209, 2008.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[TMD09] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[WB09] Craig D. Weissman and Steve Bobrowski. The Design of the Force.com Multitenant Internet Application Development Platform. In *Proc. 35th SIGMOD International Conference on Management of Data (SIGMOD '09)*, pages 889–896, New York, NY, USA, 2009. ACM.

[WGG+08] Zhi Hu Wang, Chang Jie Guo, Bo Gao, Wei Sun, Zhen Zhang, and Wen Hao An. A Study and Performance Evaluation of the Multi-Tenant Data Tier Design Patterns for Service Oriented Computing. In *Proc. Int. Conf. on E-Business Enigneering (ICEBE'08)*, pages 94–101. IEEE, 2008.

Anforderungen klären mit Videoclips

Kurt Schneider

Lehrstuhl Software Engineering
Leibniz Universität Hannover
Welfengarten 1, 30167 Hannover
Kurt.Schneider@inf.uni-hannover.de

Abstract: Viele große Softwaresysteme sind heute vernetzt, in Geräte eingebettet und mit Anzeigesystemen verbunden. Sie erscheinen den Nutzern als komplizierte, computergestützte Umwelt, deren Bestandteile kaum zu unterscheiden sind. Die Geräte und ihre Umgebung, die Software und die Bedürfnisse der Benutzer entwickeln sich während des Betriebs ständig weiter. Damit ein Systemteil nützlich und wettbewerbsfähig bleibt, braucht man fortwährend Rückmeldungen und Bewertungen. Die Techniken des klassischen Requirements Engineering reichen dazu aber nicht aus. In diesem Beitrag stellen wir einen Ansatz vor, um mit kurzen und einfachen Videoclips in dieser Situation an Feedback heranzukommen. Bei deren Auswertung werden aktuelle Anforderungen identifiziert und geklärt.

1 Einleitung

Viele technische Systeme interagieren heute eng miteinander: Navigationssysteme erhalten Stauinformationen über das Radio, der Lautsprecher wird leiser gestellt, sobald eine Verkehrsfunkanzeige empfangen wird. Im öffentlichen Raum (Bahnhof, Flughafen, Parkhäuser) haben es die Bürger als „Stakeholder“ mit immer enger verflochtenen Systemen von Systemen zu tun. Der Begriff des „IT-Ökosystem“ [NTH09] spielt auf diese Situation an, in der unabhängig entwickelte Systemteile teilweise autonom die Dienste anderer Teilsysteme in Anspruch nehmen. Stakeholder, also betroffene oder nutzende Menschen und Gruppen nehmen das Ganze als mehr oder weniger „intelligente“ Umwelt wahr.

1.1 Problemstellung: Anforderungen klären in bereits laufenden IT-Ökosystemen

Auch Geräte und Systeme in solch einer Umgebung müssen Anforderungen erfüllen. Durch die vielfältigen Interaktionen der Systeme und Stakeholder im Betrieb verändern sich diese Anforderungen ständig, ebenso wie die Umweltbedingungen. Softwareunternehmen müssten ständig die Leistungen ihrer Produkte im Gesamtsystem mit den aktuellen Anforderungen der Stakeholder abgleichen, um wettbewerbsfähig zu bleiben [Je08]. Doch dies ist aus mehreren Gründen schwierig:

- Klassisches Requirements Engineering mit Befragung von Stakeholdern in Interviews und Workshops ist äußerst aufwändig.

- Verschiedene Personengruppen und Personen können unterschiedliche Anforderungen an öffentliche Systeme haben. Für Anbieter eines softwaregestützten Systems sind dabei besonders gegenwärtige und potenzielle Kunden relevant. Einzelne aufwändig Interviewte repräsentieren kaum die ganze Zielgruppe. Eine größere Zahl von Personen einzubinden, erzeugt noch höhere Kosten für Marktforschung und Anforderungsinterviews.

- Bei der Befragung oder Validierung im Labor müssen sich Stakeholder an die Interaktion mit dem komplexen System erinnern oder sie nachvollziehen. Die Laborumgebung kann Beobachtungen verfälschen. Beispielsweise lässt sich die Verständlichkeit von Anzeigetafeln auf dem Bahnhof nur schlecht im Labor nachempfinden, weil die Umwelteinflüsse (Lärm, viele Menschen, Eile) fehlen.

- Durch die Interaktion mit neuen, autonomen Teilsystemen ändert sich auch das wahrgenommene Gesamtverhalten ständig. Ein störender Effekt ist kaum reproduzierbar, wenn er aus dem Zusammenwirken mit anderen Teilsystemen entstanden ist. Es ist unmöglich, alle Kombinationen und Interaktionen im Labor zu validieren, da die Umwelt dort nicht in gleicher Weise zur Verfügung steht.

Anforderungserhebung und -validierung in der realen Umgebung durchzuführen (z.B. im Bahnhof), erscheint daher als naheliegende Alternative. Doch bisher mussten teuere Experten die Stakeholder beobachten, und die anderen obigen Probleme blieben bestehen.

1.2 Ein neuer Ansatz zum Umgang mit Videoclips für Feedback

Software in vernetzten Systemen und IT-Ökosystemen wird nur selten ganz neu erstellt. Meist sind mehrere konkurrierende Komponenten und Produkte im Einsatz, die nach und nach verändert oder ersetzt werden. Welche davon weiter genutzt (und bezahlt) werden, hängt davon ab, wie gut und schnell die Unternehmen den Bedürfnissen ihrer Kunden und Benutzer folgen können. Requirements Engineering entwickelt sich immer mehr von einer einmaligen Entwicklungsaktivität zu einer Tätigkeit, die den ganzen Lebenszyklus begleitet - und stützt. Mit dieser Situation beschäftigen sich auch das Produktmanagement [Je08] und das market-driven requirements engineering [Ka02].

In Zukunft wird die Wettbewerbsfähigkeit von **Unternehmen** noch stärker davon abhängen, ob sie Probleme und veränderte Anforderungen an ihre laufende Software rasch erkennen und umsetzen können [Ka02]. Unternehmen brauchen Rückmeldung und Verbesserungshinweise für ihre Software und ihre Geräte. Mit der Verbreitung von Fotohandys und Flatrates sinkt die Hemmschwelle für die vorgeschlagene Art von Rückmeldungen. Je mehr Stakeholder Feedback schicken, desto größer ist auch die Chancen, ungünstige Konstellationen in komplexen Systemen „zufällig“ zu entdecken. Wenn viele Bürger einfach Rückmeldungen geben können, eröffnet dies Unternehmen einen zusätzlichen Kommunikationskanal zu ihren Kunden, was zu verbesserter Kundenbindung führen kann - wenn die Kritikpunkte dann auch tatsächlich rasch beseitigt werden. Aber auch aus Sicht vieler **Bürger** hat der vorgeschlagene Ansatz Vorteile: Viele Bürger gehen durchaus kritisch mit den Angeboten um, die sie im öffentlichen Raum vorfinden, also in Bahnhöfen, Krankenhäusern und auf Flughäfen. Technisch Interessierte nehmen Fehlver-

halten von Systemen bewusst als solches wahr und sprechen (z.B. am Bahnsteig) über verwirrende Anzeigen und fehlende Funktionen. Viele wären vermutlich gerne bereit, eine kurze Rückmeldung zu geben, wenn sie das Gefühl hätten, damit Systeme verbessern zu helfen, die sie selbst ständig nutzen. Kaum jemand nimmt sich freilich die Zeit, Probleme und Beobachtungen später von zu Hause aus zu melden. Auf Bahnhöfen, Flughäfen und in ähnlichen hochtechnisierten Umgebungen treten dagegen ohnehin oft Wartezeiten auf – sie sind oft Teil des Problems. Diese Wartezeiten bieten eine gute Chance, an Rückmeldungen zu kommen: Manche Stakeholder sind sogar froh, ihren Ärger und ihre Anregung loswerden zu können – statt nur zu warten, sich zu ärgern oder zu langweilen. Kleine Anreize können als extrinsische Motivation hinzukommen, um sinnvolle Einsendungen zu honorieren. Nach Davenport [DP00] müssen Anreize nicht finanzieller Natur oder gar teuer sein. Wichtiger sind Anerkennung und schnelle Verbesserung der gemeldeten Missstände. Frustrationsabbau, aktive Partizipation an der Verbesserung selbst genutzter Systeme sind intrinsische Motivationen vieler technologieinteressierter Bürger. Diese win-win-Situation zwischen Bürgern und Unternehmen sollte man nutzen und fördern.

In diesem Beitrag wird ein neuer Ansatz beschrieben, um mit einfachen Videoclips Rückmeldungen zu erfassen und Anforderungen zu klären. Dazu werden ein Prozess und eine Grobarchitektur vorgeschlagen. Vorarbeiten aus verschiedenen Bereichen werden als Bestandteile genutzt, und neuartig kombiniert Der Ansatz nutzt die drastisch veränderten Möglichkeiten durch Fotohandys, Flatrates und technologieaffine Bürger.

Zur Konkretisierung stellt dieser Beitrag auch ein neues Video-Bearbeitungswerkzeug vor, das in einer Masterarbeit eigens erstellt wurde, um einen wichtigen Teil des neuen Ansatzes noch gezielter zu unterstützen. Die Idee, Videos für die Anforderungsklärung einzusetzen, ist nicht neu. Kapitel 2 stellt verwandte Arbeiten in verschiedenen Bereichen vor. Kapitel 3 fasst den Ansatz zur interaktiven Verwendung von Videoclips für die Anforderungsklärung zusammen. In Kapitel 4 wird der Stand der Arbeiten zur Umsetzung und Evaluierung gezeigt und in Kap. 5 wird ein kurzes Fazit gezogen.

2 Verwandte Arbeiten

Requirements Engineering vermittelt zwischen Softwareentwicklern und verschiedenen Stakeholdern. Neben den klassischen Befragungstechniken, Interview und Workshop, werden für die Analyse komplexer Probleme und Situationen ethnographische Beobachtungsansätze im realen Umfeld empfohlen [Hu95]. Auch im Produktmanagement [Je08] und im market-driven requirements engineering [Ka02] müssen Anforderungen aus dem Markt erhoben werden. Karlsson et al. [Ka02] weisen auf die Bedeutung von Feedback und neuen Anforderungen im Umgang mit bereits existierenden Produktversionen hin.

Feedback erheben durch aktive Beteiligung: Unser Vorschlag setzt auf aktive Beteiligung der Stakeholder: Sie werden nicht beobachtet, sondern beobachten selbst. Vorbild sind Varianten des szenario-basierten Entwurfs, wie das Rollenspiel oder die so genannten „Playthroughs“ [AD02], bei denen man so tut, als hätte man ein gewünschtes Gerät bereits. Die dann mögliche Interaktion wird durchgespielt und auf diese Weise spezifiziert. Wirf-Brocks weist dabei Akteuren die Rollen“Benutzern” und sogar “System” zu [Wi95].

Perspektivenwechsel zur Anforderungsklärung: Im Sinne der Anforderungsklärung muss man versuchen, die relevanten Aspekte eines Rollenspiels oder Videoclips zu identifizieren [St73]. Dazu kann ein Perspektivwechsel nützlich sein. In ihrer Arbeit über Viewpoints haben Nuseibeh et al. diesen Punkt betont [NFK94]. Szenarien und Interaktionssequenzen sind essentielle Bestandteile vieler Spezifikationen in der Praxis [AM04, Co05]. Wie Alexander and Maiden darlegen [AM04], gibt es zahlreiche Typen von Szenarien im RE. Use cases beschreiben beispielsweise die Abfolge der Schritte eines Szenarios und fordern Zusatzinformationen wie Voraussetzung, Auslöser und Stakeholderinteressen ein [Co05].

Anforderungen ohne klare Adressaten mit Videoclips erheben: Wenn die Öffentlichkeit bzw. die Kunden Anregungen und Feedback geben sollen, dann muss die Darstellung weitgehend ohne fachspezifische Notation und Einarbeitung auskommen. Das ist bei Videoclips der Fall. Die heutige Situation unterscheidet sich zudem deutlich von 1998, als Haumer et al. [HPW98] den Einsatz von Videos für das Requirements Engineering untersucht haben. Sie schlagen vor, „rich media“ (Videos, Audio, Skizzen, Bilder) gemischt einzusetzen und auf einem Whiteboard Editor zu arrangieren. Dann können die Ergebnisse im Labor diskutiert werden. Bei Haumer et al. [HPW98] werden semi-formale Goal Models in Evaluierungssitzungen mit rich media verknüpft.

Aufwändige Visionsvideos für Produkte und Einsatzszenarien: Brügge und Creighton verwenden Visionsvideos. Diese werden in der Werbung, im Marketing und im Requirements Engineering benutzt. Ein bekannter Vertreter ist Apples Vision Video von 1987 (http://www.youtube.com/watch?v=3WdS4TscWH8). Bei Brügge und Creighton zeigt ein Visionsvideo beispielsweise, wie sich ein Hersteller die Nutzung eines Computertomographen vorstellt [COB06]. Die Visionsvideos können mit Sequenzdiagrammen angereichert werden, so dass man die relevanten Aspekte der Videos auf die formaleren Modelle abbilden kann. Man will auf diese Weise Abläufe mit „Normalbürgern“ und Stakeholdern validieren und sie dann in UML übertragen, so dass wertvolles Feedback gezielt an die Entwickler weitergegeben wird. Die professionellen Videos und ihre Erweiterung um UML-Modelle sind jedoch äußerst aufwändig und teuer. Ihre Erstellung dauert relativ lange und lohnt sich nur, wenn auch das Produkt entsprechend hohe Gewinne erwarten lässt. In [BSR08] wenden Brügge et al. ähnliche Techniken in der Software-Engineering-Lehre an, wobei wiederum aufwändige und teuere Studiotechniken, hier das “green screening”, genutzt werden. Ungeachtet des hohen Aufwands belegen die Arbeiten, dass Benutzer anhand von Videos komplexe Handlungsabläufe wirksam diskutieren können.

Leichtgewichtige Videobearbeitung: Einige Ansätze beschäftigen sich mit leichtgewichtigen Techniken für „Video-on-the-fly“: Hagedorn et al. [HHK08] stellen einen Ansatz zur Videoannotation vor, was ein Schritt in Richtung leichtgewichtiger Semantikanreicherung und Indizierung ist. Engström et al. beschreiben einen “video jockey (VJ)”, mit dem man interaktiv Videos arrangieren kann, in Analogie zu disc jockeys [EEJ08]. Videoclips eignen sich für einfaches Feedback und für Diskussionen, wenn man sie leicht rekombinieren kann. Maiden et al [Ma06] haben mobile Endgeräte eingesetzt, um zu studieren, wie man diese Geräte zur Anzeige von Checklisten und Fragen über Anforderungen verwenden kann. Wir wollen dagegen nicht Checklisten zeigen, sondern Videos mit solchen Endgeräten erfassen und gleich mit einigen semantischen Annotationen versehen.

3 Videoclips für Feedback: Idee, Prozess und Grobarchitektur

Etliche der oben angesprochenen Forscher versuchen, möglichst viel aus Videos und anderen „rich media“ herauszuholen. Sie sind bereit, dafür viel Aufwand in die Erstellung, Annotierung und Diskussion der Videos zu stecken. Alternativ kann man einfach Videos aufzunehmen und ohne jede Verarbeitung versuchen, daraus Anforderungen abzulesen. Dabei wird der Aufwand für die Videoersteller zwar niedriger, die Auswertung ist dagegen umso schwieriger.

Es gab jedoch bisher kein Konzept, wie Videoclips zwar sehr einfach erhoben, andererseits aber aussagekräftig ausgewertet werden können. Anforderungserhebung mit Fotohandys während des laufenden Betriebs erfordert eine durchgängige Idee und einige neue Komponenten. Darauf geht dieser Abschnitt ein.

3.1 Idee: Spezialwerkzeuge erleichtern Feedback und Auswertung

Unser Ansatz liegt in der Mitte zwischen den beiden oben angesprochenen Extremen: Der Aufwand für die Stakeholder soll minimiert werden. Wir investieren dafür in der Forschung relativ viel Aufwand in die Entwicklung von Werkzeugen, die sowohl die Formulierung als auch die Auswertung von Feedback erleichtern. Vollständige Erfassung aller Anregungen und Wünsche ist dagegen hier nicht das Ziel. Man sollte realistischerweise davon ausgehen, dass ein normaler Bürger nur dann ein Video erstellen und einsenden wird, wenn er sich dafür unmittelbare Vorteile verspricht und keine wichtige Tätigkeit unterbrechen muss. Solche Situationen gibt es! Rückmeldungen reagieren in der Regel auf Unterbrechungen und Störungen, wenn die beabsichtigte Tätigkeit ohnehin gestört ist (z.B. Ticket kaufen, Bahn fahren). Es ist für unseren Ansatz wichtig, solche Gelegenheiten zu identifizieren und auszunutzen.

Die Grundprinzipien unseres Ansatzes lauten:

- Die Schwelle, einen Videoclip einzusenden, muss in jeder Hinsicht niedrig sein.
- Videoclips werden nicht professionell bearbeitet und geschnitten. Vielmehr werden sie interaktiv arrangiert, wobei ein Editorwerkzeug hilft.
- Idealerweise enthalten Videoclips auch Metadaten zu Aufnahme, Autor und Absicht, sowie einigen anderen Aspekten, die in einem speziell angefertigten Editorwerkzeug für die Montage und Diskussion genutzt werden können.
- Dabei spielt der Kontext der Rückmeldung eine wichtige Rolle. Er liefert semantische Informationen, um den oder die Adressaten eines Videos zu ermitteln. Eine kurze, simple Interaktion ist dabei akzeptabel, aber nicht mehr.
- Wir nutzen in der Auswertung verschiedene Möglichkeiten, darunter die Gegenüberstellung unterschiedlich formaler Modelle: einerseits Videoclips und rich media, andererseits Use Cases oder ähnliche semi-formale Modelle.

Diese Prinzipien müssen in konkrete Handlungsabläufe (Prozesse) umgesetzt werden.

3.2 Prozess und Arbeitsschritte der Anforderungsklärung mit Videoclips

Abb. 1 zeigt einen Überblick über die wesentlichen Schritte beim Einsatz von Videoclips für die Anforderungsklärung. Stakeholder müssen zunächst überhaupt wahrnehmen, dass ein Problem vorliegt oder sie einen Verbesserungsvorschlag haben. Dann wird mit der Videokamera oder dem Handy etwas aufgezeichnet. Bevor dieses Video eingesandt werden kann, muss der geeignete Adressat identifiziert werden – an wen soll das Video also geschickt werden? Der Empfänger ist in der Regel ein Softwareunternehmen, das mit Hilfe der eingehenden Rückmeldungen einen bestimmten Systemteil überarbeitet und verbessert.

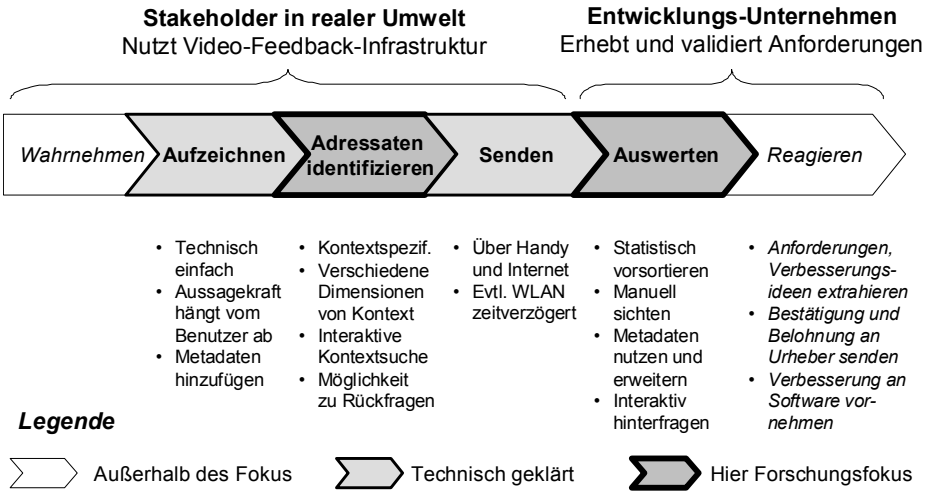


Abbildung 1: Arbeitsschritte der Anforderungsklärung mit Videoclips aus Anwendersicht

Der Forschungsfokus unserer Arbeiten liegt auf dem Gesamtablauf und darin besonders auf den hervorgehobenen beiden Schritten. In *diesem Beitrag* werden die Herausforderungen beider Schritte genannt, aber aus Platzgründen wir nur zur Auswertung auch ein eigens entwickeltes Werkzeug angesprochen.

Die Aktivitäten **Reagieren** und **Wahrnehmen** können hier nicht vertieft diskutiert werden; sie sind aber für das Funktionieren des Ansatzes unverzichtbar. Die ausgewerteten Videoclips werden als Anforderungen interpretiert und zu Change Requests umformuliert. Die ausgewählten Anregungen werden schließlich in die Software eingearbeitet. Am Ende sollte eine Bestätigung bzw. „Belohnung" der Inputgeber erfolgen. Die eigentliche **Videoaufzeichnung** sowie das **Versenden** von Fotos und Videos stellen heute weder ein technisches noch ein finanzielles Problem dar. Im Idealfall entstehen für den Rückmeldungsgeber keinerlei Kosten. Manche Kunden haben ohnehin Flatrates. Auch kann man Rufnummern einrichten, die für Anrufer kostenlos sind. Um die Bandbreitenunterschiede zu berücksichtigen, kann es sinnvoll sein, ein Video erst dann (verzögert) zu übertragen, wenn sich das Mobilgerät wieder in der Reichweite eines WLAN befindet.

3.3 Grobarchitektur für die Unterstützung der wichtigsten Schritte

Abb. 2 zeigt die Verteilung zwischen einer Zentrale, den Repräsentanten für registrierte Teilsysteme eines Anbieters, und den Endgeräten in Benutzerhand. In den registrierten Handys, Kameras und PCs sind Komponenten enthalten, die über das Internet mit der Zentrale und den Repräsentanten für Teilsysteme verbunden sind. Die Anbieter-Komponenten werden von den teilnehmenden Unternehmen betrieben und stehen ihnen für die Auswertung zur Verfügung. Der Verbesserungspfeil unten im Bild stellt die Auswirkung auf die Software in der realen Umgebung dar. Stakeholder erfahren dadurch eine Verbesserung und können auf den neuen Zustand wieder reagieren.

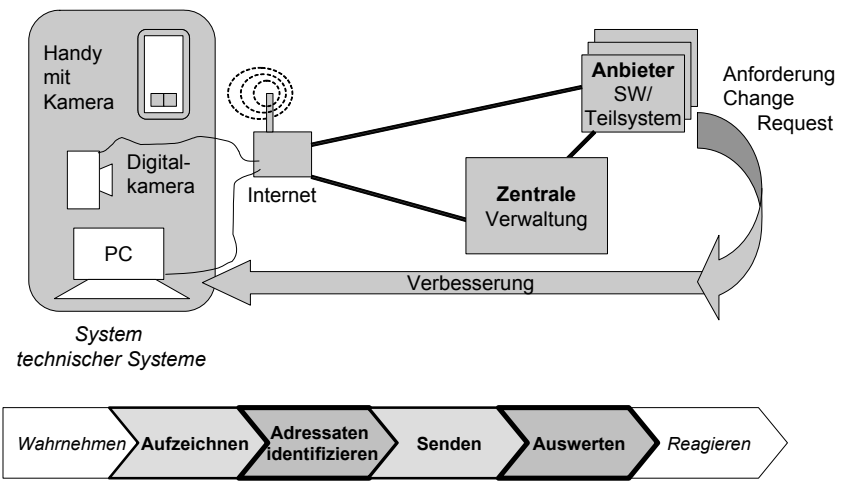


Abbildung 2: Verteilung der Arbeitsschritte auf Komponenten (Grobarchitektur)

Wenn ein Unternehmen seine Softwareprodukte bei der Zentrale registriert, wird dabei auch festgelegt, unter welchen Umständen sie als möglicher Kontext für eine Rückmeldungen betrachtet werden. So lassen sich für die Rückmeldung **Adressaten identifizieren**. Dabei spielen mehrere Dimensionen des Kontextbegriffs zusammen:

- Geographische Nähe, wenn der Stakeholder z.B. vor dem Bankautomaten steht.
- Logische Nähe, wenn im Handy gerade ein Programm geöffnet ist, z.B. bei der Online-Reservierung von Fahrkarten.
- Sender (WLAN, Bluetooth), mit denen der Stakeholder gerade interagieren kann.

Es gibt unterschiedliche Hypothesen, welchen Kontext ein Stakeholder meint, wenn er in einer gewissen Situation Feedback gibt. Stakeholder wissen oft nicht, welche Teilsysteme überhaupt existieren und ob sie für das Gesamtsystemverhalten eine Rolle spielen. Beispielsweise weiß ein Bahnkunde in der Regel nicht, wer im Bahnhof für welche Anzeigetafel zuständig ist. Dagegen kann er durchaus beantworten, ob er gerade eine Rückmeldung zur *geographisch nahen* Anzeigetafel, zur gerade *im Handybrowser*

geöffneten Online-Reservierung oder zum *physischen* Bahnhofsgebäude formulieren möchte. Bei diesem Kontext-Suchvorgang bewegt man sich heuristisch vom Speziellen, Nahen und Naheliegenden zu größeren und generischeren Kontexten, bis der Benutzer durch Tastendruck bestätigt, was er gemeint hat. Wir verwenden diese Heuristik, rechnen aber damit, sie noch verbessern zu müssen.

Das **Auswerten** der eingesandten Videoclips liefert die eigentlichen Anforderungen und Change Requests. Daher liegt diese Tätigkeit im Fokus unseres Ansatzes und dieses Beitrags. Wie oben bereits ausgeführt, sollen dazu Techniken eingesetzt werden, die die Stakeholder und die Anforderungsanalytiker wenig belasten. Einerseits müssen Gelegenheiten zum automatischen Kennzeichnen (Tagging) genutzt werden. Grundlegende Angaben (Absender, Ort, Uhrzeit, evtl. GPS-Standort) kann man technisch automatisiert ermitteln und dem Video als Tags mitgeben. Bereits mit solchen Angaben lassen sich Videos vorsortieren, gruppieren und gezielter nachbearbeiten. Zusätzlich ist es wichtig, Videos mit Audio-Erläuterungen zu versehen: Ein Bahnkunde kann eine defekte Anzeige filmen - und sollte dabei erklären, was daran gerade auffällig und besonders problematisch für ihn ist. Aus diesen Audioannotationen kann man später über die automatisch vergebenen Tags hinaus semantische Angaben extrahieren. Weitergehende Auswertungen sollen durch Spezialwerkzeuge ermöglicht werden; verwandte Beispiele sind [Sc07, Sc08].

4 Stand der Umsetzung

Der wichtigste Beitrag dieses Artikels besteht darin, den Ansatz für einfaches Feedback mit Videoclips vorzustellen. Derzeit sind verschiedene Schritte des Ansatzes in Arbeit. Abb. 3 zeigt noch einmal die Schritte des Ansatzes und gibt an, an welchen Aspekten derzeit an der Leibniz Universität Hannover gearbeitet wird. Dazu gehören auch erste Versuche, den Ansatz zu evaluieren. Eine ausführliche Evaluierung soll sich anschließen. Die annotierten Angaben werden nun anhand von Abb. 3 von links nach rechts erläutert. Zuletzt wird der Spezialeditor etwas ausführlicher dargestellt.

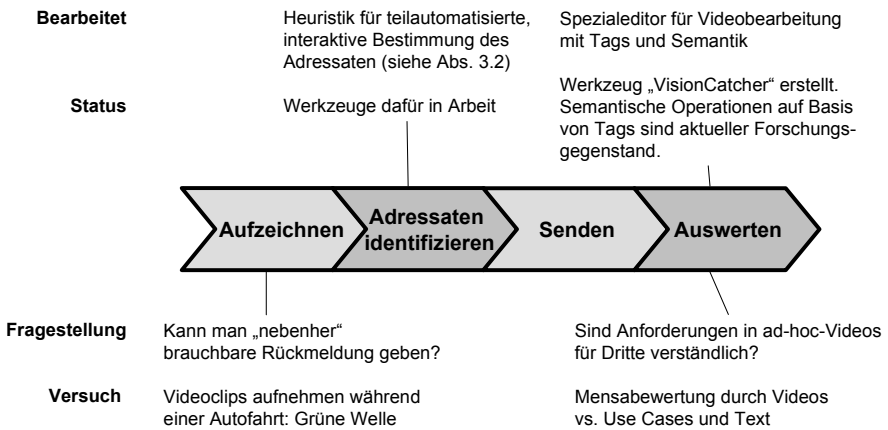


Abbildung 3: Übersicht über laufende Arbeiten, Status und Evaluierungsaktivitäten

Kann man „nebenher“ brauchbare Rückmeldung geben? Hierfür wurden kurze Videoclips während einer Autofahrt aufgenommen. Aus der Position des Beifahrers wurden Fahrsituationen, der Fahrer selbst und ein fiktives, erwünschtes Gerät gefilmt. Teile davon sind in Abb. 4 zu sehen. Besonders beim Warten an Ampeln hat der Fahrer Kritik und Wünsche gesprochen, während gefilmt wurde. Während der ca. 20-minütigen Fahrt entstanden 18 kurze Videoclips. Das Video wurde in weniger als einer Stunde mit Mockups eines „Grüne-Welle-Assistenten “ kombiniert und dann bei verschiedenen Gelegenheiten gezeigt. Die Zuschauer konnten die Intention erkennen, wiedergeben und sie waren in der Lage, Alternativen oder weitere Anforderungen für den Assistenten zu formulieren.

Heuristik für halbautomatisierte Identifikation von Adressaten: Die Videoclips werden mit handelsüblichen videofähigen Handys (oder Digitalkameras) aufgenommen. Derzeit laufen studentische Softwareprojekte, in denen die teils automatisierte, teils interaktive Identifikation des besten Adressaten für eine Rückmeldung implementiert wird. Dafür stehen Android-Handys mit integrierter Videofunktion zur Verfügung. In diesem Schritt muss die Architektur aus Abb. 2 konkretisiert und eingesetzt werden. Der identifizierte Adressat, also das angesprochene Teilsystem aus Abb. 2, soll bis zu drei kurze Rückfragen stellen können, wenn es Feedback erhält. Sie sollen möglichst durch einfache Auswahl oder ja/nein zu beantworten sein, um den Aufwand für den Feedback-Geber gering zu halten. Durch solche vorstrukturierten Zusatzangaben ist es möglich, große Mengen von Feedback zu kanalisieren, sie statistisch vorauszuwerten und auch die Videoclips automatisch mit Anmerkungen bzw. Tags zu versehen.

Sind Anforderungen in einem ad hoc-Video für Dritte verständlich? Eine Person erklärte einer Gruppe von vier Doktoranden die Idee eines „Mensa-Bewertungssystems“. Dann sollten zwei der Doktoranden das System mit Use Cases und Text beschreiben, die beiden anderen sollten mit Hilfe einer Videokamera ein kurzes Video drehen. Dafür standen jeweils 30 Minuten zur Verfügung, anschließend noch 15 Minuten für den Schnitt auf dem Storyboard bzw. Ausformulieren der Texte. Zwei Softwareentwickler (als Empfänger der Anforderungen) und zwei Studierende, die oft in die Mensa gehen (als Stakeholder) wurden unabhängig gebeten, auf der Basis des Videos bzw. der Use Cases wesentliche Eigenschaften des beschriebenen Systems zu identifizieren (speak-aloud). Dann wurden noch Fragen gestellt. Die wichtigsten 13 Anforderungen waren zuvor schriftlich festgehalten worden, um als Referenz zu dienen. Die Zahl der erkannten Anforderungen lag in beiden Fällen bei knapp über der Hälfte, was für die kurze Erstellungszeit akzeptabel ist. Das bestätigt, dass auch sehr kurzfristig und schnell erstellte Videos wirksam Anforderungen transportieren können. Wie zu erwarten, konnten Entwickler mehr Anforderungen identifizieren als Stakeholder; Entwickler haben das gelernt. Im subjektiven Urteil schnitten die Videos bei den Entwicklern erstaunlicherweise sogar besser ab als bei den Stakeholdern.

Spezialeditor für Videobearbeitung mit Tags und Semantik. Eine Herausforderung bei der Auswertung besteht darin, einerseits den Aufwand für alle Beteiligten gering zu halten, und andererseits doch hilfreiche Schlüsse aus den Videoclips zu ziehen.

Zu diesem Zweck sollten ad-hoc-Videos mit Annotationen versehen und während eines Interviews validiert und nachbearbeitet werden. Dazu wurde eine Masterarbeit definiert,

in der die Kernfunktionalität für Videoclip-Kombination mit Funktionen zum Annotieren („Tagging“) integriert werden sollte. Kitzmanns rich media-Mischpult mit dem Namen „VisionCatcher“ greift die Idee des Whiteboard Editors auf [HPW98]. Verschiedene Medien (Videoclips, Fotos, Handskizzen, Audio usw.) können einfach eingelesen und neu angeordnet werden. Längere Videos werden zunächst in kurze Clips zerlegt. So lassen sich leichter Varianten kombinieren und vergleichen. In Interview und Auswertung arbeiten Softwarefachleute des Unternehmens und können bei Bedarf einzelne Kunden gezielt zu ihrer Interpretation und ihren Präferenzen befragen. Dazu braucht man viel weniger Zeit als bei einer „Schrotschuss-Befragung“ ohne vorherigen Videoinput.

Kitzmanns Mischpult soll auf einem TabletPC während der Interviews eingesetzt werden. Es ähnelt auf den ersten Blick kommerziellen Storyboards (Abb. 4), Bedienbarkeit der Hauptfunktionen war hier aber wichtiger als ein großer Funktionsumfang. Links ist die Palette von Videoclips, Bildern usw. zu sehen. Durch drag-and-drop kann man diese Elemente auf den Filmstreifen (im unteren Bildteil) ziehen. Den ganzen Filmstreifen kann man sofort abspielen: Mit der Play-Taste oben startet man den Film, der sich aus den Clips, Skizzen und anderen Elementen zusammensetzt. Standbilder sind eine vordefinierte Zeit lang zu sehen. Der Film wird rechts im Bild angezeigt. In Abb. 4 läuft gerade der mittlere Videoclip. Er ist im Filmstreifen „aufgeklappt“ und kann bearbeitet werden:

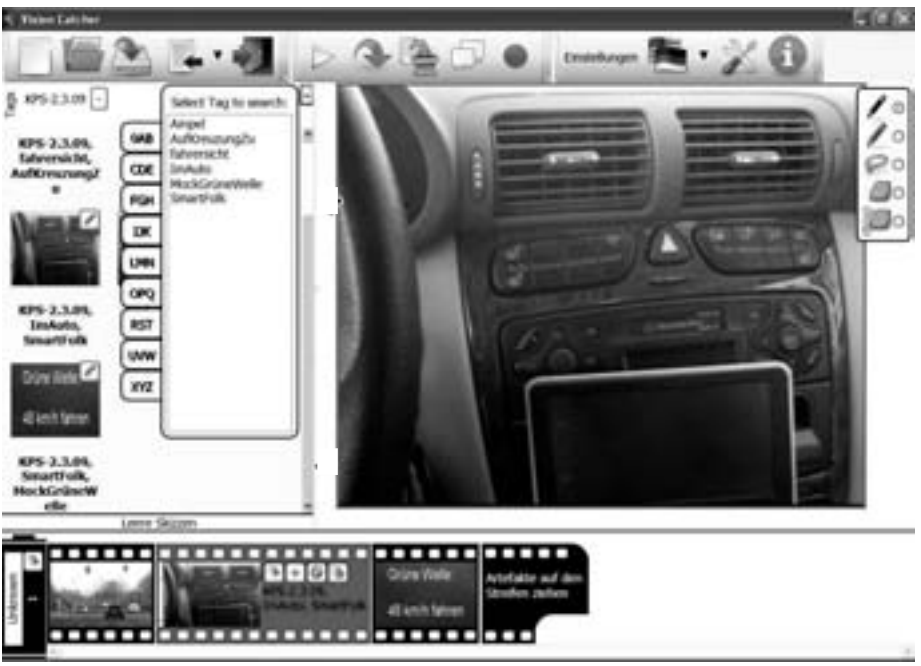


Abbildung 4: Der VisionCatcher von Kitzmann [Ki09] für Videoclipbearbeitung mit Tags

Bei der Auswertung entwickeln sich häufig interessante Diskussionen. Diese kann man über ein Mikrofon aufzeichnen (siehe runder Aufnahmeknopf oben). Außerdem kann man am TabletPC jederzeit mit dem Stift (rechts) Annotationen auf allen gezeigten Elementen

vornehmen, also „dazu schreiben“. Sie werden mit den Diskussionsaufnahmen und dem Film zusammen gespeichert. Dies fördert Interviews und anschließende Auswertung. Schon damit ist VisionCatcher ein Spezialwerkzeug zur Unterstützung des hier vorgestellten Ansatzes. Kitzmanns Arbeit [Ki09] illustriert darüber hinaus, wie semantische Annotationen (Tags) in diesem Prozess eingesetzt werden können. In Abb. 4 ist gerade die Filterfunktion aktiviert, das zugehörige Teilfenster überdeckt teilweise die Palette. Die Palette zeigt nur solche Elemente, die zu den gewählten Tags passen. Tags stehen unter den Palettenelementen.

Gerade ist ein administrativer Tag „KPS-2.3.09“ (Absender-ID und Datum) ausgewählt. Nur Elemente mit diesem Tag werden in der Palette angezeigt. Die Palette kann durch Auswahl weiterer Tags weiter eingeschränkt werden. Besonders wertvoll sind in diesem Arbeitsschritt auch explizite Audiokommentare. Der Fahrer hat im obigen Beispiel beim Warten an der roten Ampel mündlich erklärt, wieso er hier eine Grüne Welle erwartet hätte. Relevante Kommentare kann man manuell in Tags umsetzen.

Das dritte Element im Filmstreifen („Grüne Welle“) ist ein Mockup, also eine Skizze, mit dem der Interviewer eine Lösung für die Grüne Welle vorschlägt: Der Assistent im Auto empfiehlt die Geschwindigkeit, bei der auch die nächste Ampel bei Grün erreicht wird. Im Interview können Stakeholder zu diesem Vorschlag Stellung beziehen. Dieses Beispiel stammt aus dem Forschungsprojekt „IT-Ökosysteme“ [NTH09], in dem der hier beschriebene Ansatz zur Kontrolle eng vermaschter IT-Systeme verwendet werden soll.

5 Schlussfolgerungen

Videos wurden schon früher im Requirements Engineering eingesetzt. Der hier vorgestellte Ansatz geht aber von einer veränderten Situation in der Gesellschaft aus und nutzt sie. Handys mit Videokamera und Flatrates haben viele technikinteressierte Bürger zu ernstzunehmenden Stakeholdern bei der Verbesserung von Softwaresystemen gemacht.

Ein leichtgewichtiger und durch Werkzeuge gut unterstützter Ablauf wird vorgeschlagen. In diesem Beitrag werden die Schritte identifiziert und diskutiert, die zur Klärung von Anforderungen durch Videoclips führen. Werkzeuge sind dazu erforderlich. Daher wird eine Grobarchitektur für die Kommunikation vorgestellt und ein speziell entwickeltes Werkzeug zur Auswertung gezeigt. Kitzmanns VisionCatcher ist in einer Masterarbeit an der Leibniz Universität Hannover entwickelt worden.

In diesem Beitrag sollte die Vision und der Ansatz von einfachen Rückmeldungen und dennoch reichen Auswertungsmöglichkeiten gezeigt und anhand eines speziell dafür entwickelten Werkzeugs exemplarisch konkretisiert werden. Dadurch wird aber auch deutlich, dass noch erhebliches Forschungspotenzial besteht, um den Ansatz in allen Schritten umzusetzen und auszubauen. Die Herausforderungen reichen von verbesserten Heuristiken für die Identifikation von Kontext und Adressaten über intelligenteres Tagging bis hin zu semantisch reicheren Auswertungsmechanismen. Auch statistische Auswertung von Videoclips mit Tags bieten sich an. An diesen Aspekten werden wir weiterarbeiten.

Literaturverzeichnis

[AM04] Alexander, I.F. and N. Maiden, eds. Scenarios, Stories, Use Cases. Through the Systems Development Life-Cycle. 2004, John Wiley&Sons: Chichester, England.

[AD02] Austin, R., Devin, L.: Beyond requirements: software making as art. IEEE Software, 2002. 19.

[BSR08] Brügge, B., Stangl, H., Reiss, M.: An Experiment in Teaching Innovation in Software Engineering. in OOPSLA' 08. 2008. Nashville, Tennessee, USA.

[Co05] Cockburn, A.: Writing Effective Use Cases. 14th Printing ed. 2005: Addison-Wesley.

[COB06] Creighton, O., Ott, M., Brügge, B.: Software Cinema: Video-based Requirements Engineering. in 14th IEEE Internat. Requirements Engineering Conference. 2006. Minneapolis/St. Paul, Minnesota, USA.

[DP00] Davenport, T., Probst, G.: Knowledge Management Case Book - Best Practises. 2000, München, Germany: Publicis MCD, John Wiley & Sons. 260.

[EEJ08] Engström, A., Esbjörnsson, M., Juhlin, O.: Mobile Collaborative Live Video Mixing. in MobileHCI 2008. 2008. Amsterdam, the Netherlands.

[HHK08] Hagedorn, J., Hailpern, J., Karahalios, K.: VCode and VData: Illustrating a new Framework for Supporting the Video Annotation Work. in AVI 08. 2008. Naples, Italy.

[HPW98] Haumer, P., Pohl, K., Weidenhaupt, K.: Requirements Elicitation and Validation with Real World Scenes. IEEE Transactions on Software Engineering, 1998. 24(12): p. 1036-1054.

[Hu95] Hughes, J., O'Brian, J., Rodden, T., Rouncefield, M., Sommerville, I.: Presenting ethnography in the requirements process. in Second IEEE International Symposium on Requirements Engineering, March 27 - 29, 1995. 1995. York, England: IEEE Computer Society.

[Je08] Jesse, S.:Tutorium: Erhebung von Produktanforderungen durch den Requirements Engineer. in Industrialisierung des Software-Managements. 2008.

[Ka02] Karlsson, L., Dahlstedt, A., Natt och Dag, J., Regnell, B., Persson, A.: Challenges in Market-Driven Requirements Engineering - an Industrial Interview Study. in Eighth International Workshop on Requirements Engineering: Foundation for Software Quality. 2002. Sept 9-10th, 2002 Essen Germany.

[Ki09] Kitzmann, I.: Konzept und Implementierung eines Werkzeugs für multimediale Anforderungserhebung und -validierung, in Masterarbeit, FG Software Engineering. 2009, Leibniz Universität Hannover.

[Ma06] Maiden, N., Seyff, N., Grünbacher, P., Otojare, O., Mitteregger, K.: Making Mobile Requirements Engineering Tools Usable and Useful. in Requirements Engineering Conf., 2006. RE 2006. 14th IEEE International. 2006. Minneapolis.

[NTH09] NTH: IT Ökosysteme: Autonomie und Beherrschbarkeit Software-intensiver Systeme - Projekt der Niedersächsischen Technischen Hochschule (NTH). 2009.

[NKF94] Nuseibeh, B., J. Kramer, and A. Finkelstein: A framework for expressing the relationships between multiple views in requirements specification. lEEE Transactions on Software Engineering, 1994. 20(10): p. 760-773.

[Sc07] Schneider, K.: Generating Fast Feedback in Requirements Elicitation. in Requirements Engineering: Foundation for Software Quality (REFSQ 2007). 2007. Trondheim, Norway.

[Sc08] Schneider, K.: Improving Feedback on Requirements through Heuristics. in 4th World Congress for Software Quality (4WCSQ). 2008. Bethesda, Maryland, USA.

[St73] Stachowiak, H.: Allgemeine Modelltheorie. 1973, Wien, New York: Springer Verlag.

[Wi95] Wirfs-Brock, R.: Designing Scenarios: making the case for a use case framework. The Smalltalk Report, 1995. 3 (3): p. 7-10.

Indicator-Based Inspections: A Risk-Oriented Quality Assurance Approach for Dependable Systems

Frank Elberzhager, Robert Eschbach, Johannes Kloos

Testing and Inspections Department
Fraunhofer Institute for Experimental Software Engineering
Fraunhofer Platz 1
67663 Kaiserslautern, Germany
{frank.elberzhager, robert.eschbach, johannes.kloos}@iese.fraunhofer.de

Abstract: We are surrounded by ever more dependable systems, such us driving assistance systems from the automotive domain or life-supporting systems from the medical domain. Due to their increasing complexity, not only the development of but also the quality assurance for such systems are becoming increasingly difficult. They may cause various degrees of harm to their environment. Hence, in order to reduce risks associated with these systems, development as well as quality assurance normally use risk analysis as a basis for constructive and analytical measures against these risks. One of the aims of quality assurance is fault detection and fault forecasting. In this paper, the authors present indicator-based inspections using Goal Indicator Trees, a novel risk-oriented quality assurance approach for fault detection. It can be used to detect faults of different types, like safety faults or security faults. Starting from typical risk analysis results like FMECA and FTA, the approach systematically derives quality goals and refines these goals into concrete quality indicators that guide the indicator-based inspection. Quality indicators can be mapped to concrete checklists and concrete inspection goals in order to support inspectors checking artifacts in a fine-grained way with respect to certain quality properties. The approach is explained and demonstrated with respect to the quality property safety, but tends to be generalizable to further quality properties.

1 Introduction

With the growing importance of systems that help control critical aspects of our environment, such as automotive driving assistants and medical life support systems, the importance of guaranteeing the dependability of software and embedded systems is also increasing steadily. As these systems often have high inherent complexity, both constructive measures and analytical quality assurance measures are necessary to ensure the required level of dependability.

To ensure that the dependability properties required of a system are fulfilled, a wealth of measures are taken, starting with risk analysis to identify possible problems, followed by constructive measures to ensure fault avoidance and fault tolerance, and concluding with quality assurance measures for fault removal and fault forecasting. Sadly, the results derived during risk analysis are often too coarse to allow a truly satisfying quality

assurance: The descriptions of different risks and countermeasures may lack the detail needed to effectively ensure that appropriate risk reduction and prevention steps have been taken.

In particular, common risk analyses like Failure Mode, Effect and Criticality Analysis (FMECA) and cause-effect analyses like Fault Tree Analysis (FTA) tend to terminate with fairly abstract causes for hazards, and the measures taken inside the system might not have obvious connections with the hazard causes. Moreover, existing inspection reading techniques like checklists or scenarios that support inspectors in finding problems often also contain questions that are not focused enough. One method that aims to solve this problem are Security Goal Indicator Trees (SGITs [19]), which allow the refinement of security goals into indicators that can be used directly to drive inspections of the system under consideration. On the other hand, this technique is specially tailored to the security domain, not considering other quality properties such as safety or reliability.

The approach presented here builds upon the idea underlying SGITs, generalizing them into Goal Indicator Trees (GITs) so that they may be applied to different quality properties. In particular, quality goals are identified, which are then refined into sub-goals making up a particular goal. This refinement is continued until one can derive indicators, i.e., statements about the system that can be directly checked by looking at one or more system artifacts. These indicators are then used to drive an inspection, with the benefit of supporting an inspector by presenting information how to read artifacts such as requirements, design or code documents in a detailed manner. As the construction of the indicators is a systematic process in which domain experts should be involved, it is reasonably certain that all important facets for a given quality goal are considered inside a GIT. The work presented in this article describes work in progress and presents the conceptual framework towards risk-oriented indicator-based inspections demonstrated with a safety-critical example from the automotive domain.

The remainder of this work is structured into three sections: Section 2 gives an overview of related work, both with regard to analysis techniques and inspections. Section 3 describes the approach in detail, illustrating it with an example from the automotive domain. The quality property under consideration is safety, and the derivation of questions guiding an inspection for a given safety property is shown, together with the actual inspection results. Finally, Section 4 summarizes the results and gives details about ongoing evaluation activities and further research directions.

2 Related Work

Two areas should be sketched: on the one hand, methods that support analysts and developers during certain development steps ensuring a certain quality constructively and, on the other hand, those that do so analytically. Our focus is on static quality assurance methods, which are suitable for analyzing certain development artifacts such as requirements documents, design documents, models, or code.

A common classification of the means for ensuring dependability is described in [12]: A method may focus on fault avoidance, fault tolerance, fault removal, or fault prediction. Construction-time methods tend to fall into the fault avoidance and fault tolerance classes. For many of these approaches, the first step is to carry out an analysis that will identify the risks inherent in the system, as well as possible causes that lead to the manifestation of these risks. From these analyses, possible countermeasures are derived that will help to either reduce the probability of occurrence of each hazard or its severity.

There are numerous examples of such analysis techniques with different specializations. In the context of safety, two of the most common methods, often used in tandem, are FMECA and related techniques like FMEA [13][14][15] and FTA [16][22]. Both techniques have complementary goals: While FMECA is used to identify failure modes of the system and assess their criticality; FTA is used to identify the underlying causes that may lead to the manifestation of a failure mode. These techniques are usually employed together: possible system failure modes are determined using FMECA, and iteratively refined to "basic events" using FTA. These basic events are defined as "component failures" [22]. For software, this means that basic events are formulated as "software component failure", without describing possible causes.

For reliability analysis and identification of reliability-critical components, Reliability Block Diagrams (RBDs, [17]) are used. These models are built from information about the system structure and individual component reliabilities. Component failure causes are not considered.

Finally, for security, attack trees [18] allow an approach similar to FTA, while SGITs [19] are used for fine-grained analysis. From our point of view, only SGITs have a sufficient level of detail for guiding inspectors when focusing on a certain quality property: The combination of FMECA and FTA stops at a level where the failure of individual components is considered, not going into the detailed analysis of failure causes of software, while RBDs do not consider failure causes at all.

For checking certain development artifacts analytically, software inspection is a suitable means. Inspections have existed for over thirty years and were first published in 1976 by Fagan [1]. They are a structured and well-defined static quality assurance method for verifying the quality properties of different artifacts. Two main research directions can be identified regarding inspections, namely, the inspection process itself (e.g., phased inspection [3], n-fold inspection [4]) and reading support for defect detection by inspectors.

While focusing on support for developers, analysts or, inspectors in general, in order to check certain qualities, different reading support were developed that can and should be used to find defects (which can also mean violations regarding a certain quality) in an effective manner. Three different kinds can be distinguished. First, ad-hoc reading, where an inspector does not get any reading support. In this case, the inspector can perform the defect detection based solely on his knowledge and experience. Second, checklists can be used. Beside one general checklist used by each inspector [5], focused checklists [6] and guided checklists [7] were developed in order to present different

perspectives or defect classes that can be looked for in an inspection. The main advantage of focused and guided checklists is the higher defect coverage within the artifact to be checked and the lower overlap of defects found by the inspectors, i.e., inspectors mainly find different defects, resulting in higher effectiveness. One main problem with checklists is that the checklist questions are often too general, as stated in [8].

The third class of reading techniques are scenarios. The idea is that an inspector should work actively with a document instead of only reading checklist questions in a passive manner. For this, an inspector gets, for example, a description of his perspective (perspective-based reading [5]) or a certain defect class (defect-based reading [9]), which sets the focus. Next, concrete instructions have to be followed and questions have to be answered. For example, imagine an inspector taking the tester perspective. One instruction might be, "Derive a number of test cases from the corresponding document", and a possible question is, "Is all necessary information for deriving test cases stated?"

Despite many research activities regarding reading support and a number of existing reading techniques as mentioned above, little work has been done on how to support inspectors with detailed reading support to ensure certain qualities in a holistic way. The already mentioned SGITs and guided checklists are initial methods that go into this direction. Nevertheless, they only focus on the quality property security and have not been generalized to further qualities. Finally, little support is given today that describes how to derive reading support for inspectors.

In summary, despite a lot of existing constructive and analytical methods to ensure certain qualities, a generalized approach for focusing on the details of quality attributes that should be ensured across the software development cycle (i.e., in the created documents and the code) is still missing.

3 The approach

The approach for creating reading support and performing the indicator-based inspection is carried out in three steps, shown in Figure 1. To describe these steps, consider the following example: The object under consideration is the control unit for an electrically-powered car window, as described in [10]. As it is a safety-critical system, the quality assurance of this control unit must demonstrate that all safety-related non-acceptable risks have been reduced to an appropriate level [11][20].

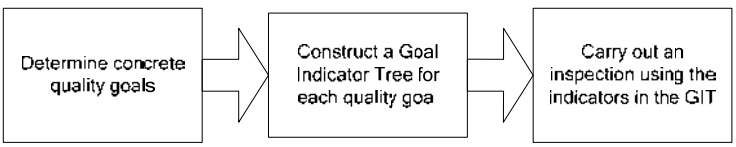


Figure 1: Steps of an indicator-based inspection

The system under consideration is called “Automotive Power Window System”. It describes an electronic control unit for an electrically controllable car door window. This control unit allows the driver (and the passenger) to move the window inside the car door using an up/down button. Additionally, it will detect whether the window is jammed by some object, using this information to avoid crushing that object. The inspection will be carried out on the requirements for this device as well as on a Matlab/Simulink model of the system (i.e., a model inspection as required in [11] for a software unit design and implementation).

3.1 Determination of concrete quality goals

The first step is the selection of the quality goals in question. For safety, the quality goals tend to be highly application-specific, since every system comes with a different set of potential safety hazards. In the automotive domain, the applicable standard (ISO WD 26262, [11]) prescribes that a safety risk analysis must be undertaken. In particular, the system’s safety hazards are to be identified, e.g., using FMECA, and the causes of each failure mode must be determined, e.g., by FTA.

Using a Fault Tree Analysis or similar techniques, one arrives at a description of all those situations in which a safety hazard may manifest itself. These situations are often described by so-called cut sets, e.g., sets of events that must occur together. One common analysis performed during FTA is the determination of minimal cut sets, e.g. of those cut sets that describe the minimum preconditions for the occurrence of a safety hazard.

For the power window controller, an FMECA includes (among others) the following failure modes:

Table 1: FMECA of power window controller (excerpt). See [14] for a description of the columns.

Error location	*Potential error*	*Consequences of error*	*S*	*CRIT*	*Causes*	*O*	*Detection*	*D*	*RPN*
Jam prevention	Window is jammed while motor moves it up	Object crushed	9	yes	Jam detection fails	4	No simple direct solution	9	72

This failure mode is hence highly safety-critical and must be further analyzed. By carrying out a Fault Tree Analysis, one arrives at the fault tree depicted in Figure 2. The software-related basic events are marked by stripes.

To ensure that a system is safe, one therefore considers each critical minimal cut set in turn, formulating a quality assurance strategy that demonstrates, for each minimal cut set, that not all its events can occur together. Thus, the concrete quality goals can be given in the form “not all events of the minimal cut set C may occur simultaneously”, where C ranges over all minimal cut sets. The minimal cut sets of the power window controller all have the form “object jams window”, “‘window up’ button pressed”, and

one of the statements “sensor does not notice jam”, “motor moves on its own”, “sensor data not received”, “sensor data misinterpreted”, and “motor still ordered to move up”.

Hence, for deriving indicators, three minimal cut sets need to be considered:

1. “object jams window”, “‘window up’ button pressed”, “sensor data not received”

2. “object jams window”, “‘window up’ button pressed”, “sensor data misinterpreted”

3. “object jams window”, “‘window up’ button pressed”, “motor still ordered to move up”

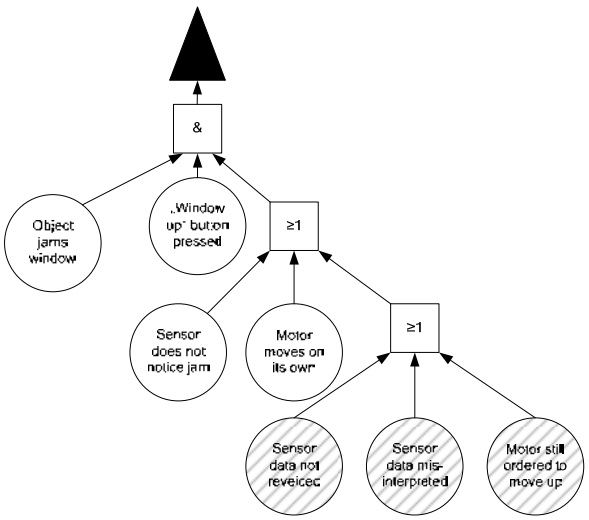


Figure 2: The fault tree for “Object jam detection fails”

From these minimal cut sets, one derives three goals:

1. When an object potentially jams the window while the “window up” button is pressed, the data from the jam sensor must be correctly received correctly.

2. When an object potentially jams the window while the “window up” button is pressed, data received from the jam sensor must be evaluated and interpreted to detect possible jamming conditions.

3. While the “window up” button is pressed, the motor must not be ordered to move up when a jam is detected.

3.2 Construction of Goal Indicator Trees for the quality goals

Next, the goals can then be iteratively refined into sub-goals and indicators. This refinement process is described exemplarily on goal 1, using a graphical notation. The result is shown in Figure 3. The detailed process and possible variations will be described in an upcoming paper; here, one variant, based on a goal refinement procedure, is illustrated on an example. A graphical notation was chosen as many common approaches (e.g., FTA and SGIT) for similar analyses also use a graphical representation.

The Goal Indicator Tree is built up from several elements, namely a single goal (at the top), two sub-goals below it, and several indicators. Goals, sub-goals, and indicators are connected by logical junctures (namely, "and", and "or"). If the goal is made up of sub-goals and indicators connected by "and", all of the sub-goals and indicators must be fulfilled for the goal to be satisfied. In the same way, if they are connected by an "or" node, at least one of them must hold for the goal to be fulfilled. The same holds for sub-goals and indicators made up of other elements. The difference between a sub-goal and an indicator is that a sub-goal may still be unspecific, while an indicator must be detailed enough to be directly checkable.

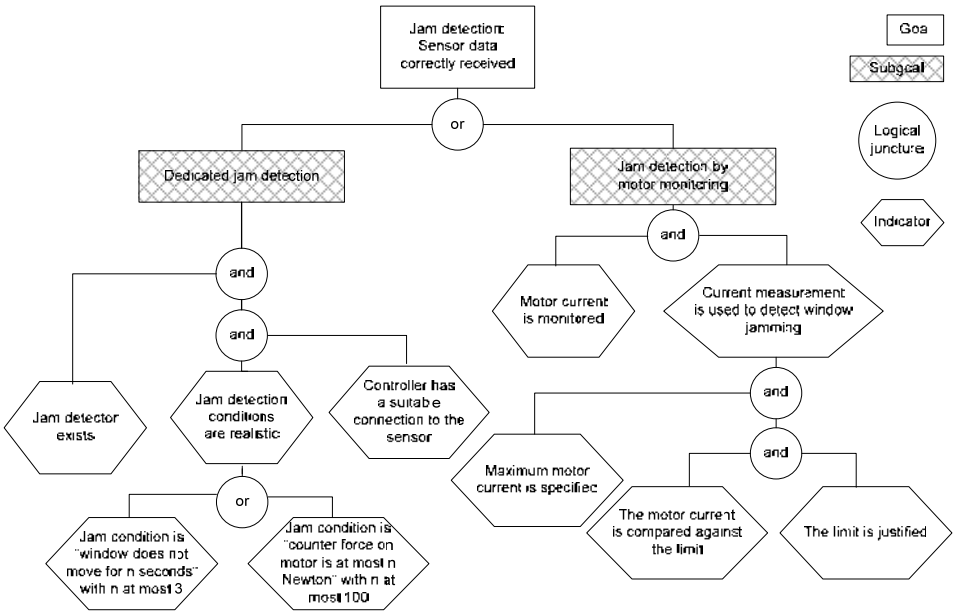


Figure 3: Derivation of indicators for a safety goal, using a Goal Indicator Tree

In the example, one starts with the first goal, "Jam detection: Sensor data correctly received". Since there are a number of possible implementations, the tree needs to be refined to allow an easier inspection. In this case, the goal describes the correctness of a system behavior that depends on the implementation of the underlying component. Thus, it is refined using an "or" juncture, resolving the dependency on the implementation by

enumeration. One arrives at two sub-goals, “Dedicated jam detection: Sensor data correctly received” and “Jam detection by motor monitoring: Sensor data correctly received”. Each of these goals describes the correctness of a system which consists of more than one component. Thus, an “and” juncture is used to split up the goal into sub-goals for each component. Following the second path, one finds that there must be both a way to monitor the motor current, and the current measurement must be used for jam detection. The existence of a motor current monitor can be directly checked by inspection (e.g., by checking whether the system has appropriate hardware). Therefore, this sub-goal is formulated as an indicator. In theory, the correct use of the current measurement could also be checked directly; hence, it is also marked as an indicator. In practice, not every inspector will be familiar with the necessary knowledge in electrical engineering to decide what an appropriate check is. Thus, this indicator is further refined.

For the second and third goal, the respective Goal Indicator Trees are shown in Figure 4.

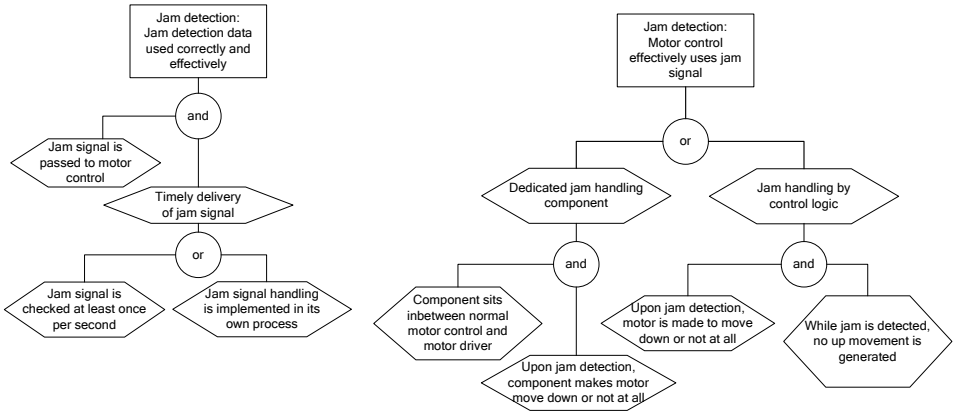


Figure 4: Goal Indicator Trees for the remaining two minimal cut sets

To keep the set of indicators manageable, one next defines assessment and selection criteria for indicators. For safety indicators based on cut sets, the following criterion can be used: An indicator is considered useful if it significantly contributes to the demonstration that the events in one of the most critical cut sets can only occur with low probability. A cut set is critical if it has a high risk priority number. As all minimal cut sets in this example have the same risk priority number, each of them must be considered.

3.3 Carrying out the inspection

Based on the defined Goal Indicator Tree shown in Figure 3, the inspection of the corresponding development documents is performed in order to find problems that violate the goal “Jam detection: Sensor data correctly received”. With respect to the inspection process, the focus is on the defect detection phase and there is a description of how to apply the GIT. A requirements document (Simulink – Automotive Power

Window System Demo – Part 1 – Designing the Controller) and a Matlab Simulink model ("Power Window", as linked from the requirements document) are used as artifacts to be analyzed, i.e., each indicator should be checked against these artifacts [10]. The goal of an inspection is to find all potential defects. Thus, it is important to check and document each indicator to the extent possible in order to find all defects that do not comply with the indicators. The order of checking is depth-first, from left to right. Finally, the GIT presents for each indicator only a short description of what to check. This explanation is often not detailed enough for non-experts regarding certain goals or for people using a GIT the first time. Thus, it is possible to derive a checklist containing more details for each indicator. Basically, each indicator is transformed into one question and enhanced with more information about what to check and how to proceed [7]. This step is skipped here. Nevertheless, in the following, more detailed questions than seen in the GIT are presented exemplarily in order to clarify some indicators.

Starting with the goal as entry point, the focus of the GIT to be checked is given. Following the described order, the sub-goal "Dedicated jam detection" has to be checked initially. The first indicator leads to the question, "Does a dedicated piece of hardware exist for jam detection?" This can be checked both in the requirements and in the model. The requirements document presents information about how an obstacle is treated while the window is closing. No dedicated piece of hardware is defined that violates the indicator. Consequently, the indicators below, which describe how to analyze implementation details, cannot be checked because the implementation of the jam detection is not done this way. Furthermore, the indicator that is not fulfilled corrupts the sub-goal and a potential problem is found (which should be documented).

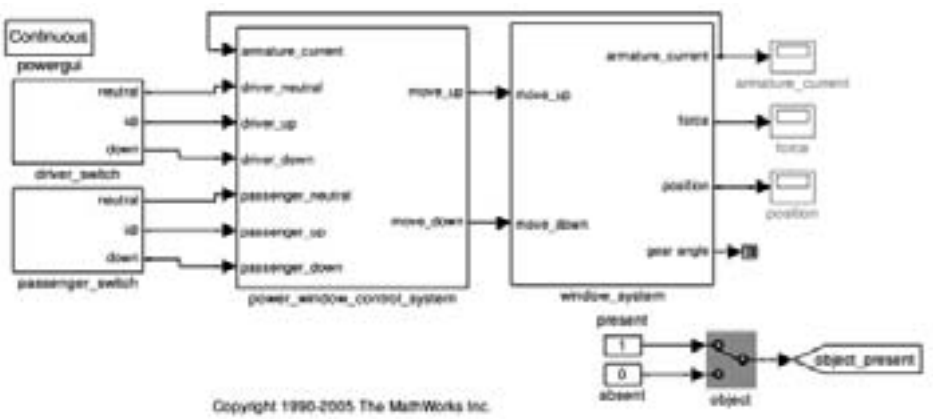


Figure 5: A high-level view of the power window control system, showing that the current running through the window movement motor is measured (the line between the armature_current connectors, at the top of the model).

Next, the second sub-goal "Jam detection by motor monitoring" should be ensured. For this, the first indicator to be checked leads to the question of whether the motor current is monitored. In the requirements document, it might say that "… the control switches to its emergency operation when a current is detected that is less than -2.5 [A]"; thus,

monitoring is documented. Next, a look is taken at the model in order to check the implementation and find out where the implementation of the requirement is done in the model (see Figure 5). A line from the window_system to the power_window_control_system exists for armature_current which indicates that the current measurement takes place.

The next indicator "Current measurement is used to detect window jamming" is checked in the requirements and could be found in a section where a description of the power window control process is given. Finally, three dependent indicators are checked, starting with the indicator "Maximum motor current is specified". Within the mentioned description, certain values for normal operation and for when an obstacle is detected are defined. That the motor current is compared against the limit can be checked in the model where the (absolute value of the) armature current measurement, whose value is read from input 2, is compared to the constant 1.2 in the comparator named "object" (see marked rectangle in Figure 6). For the last indicator, "the limit is justified", a positive answer can again be found in the requirements where the values for normal use and deviations are defined and explained.

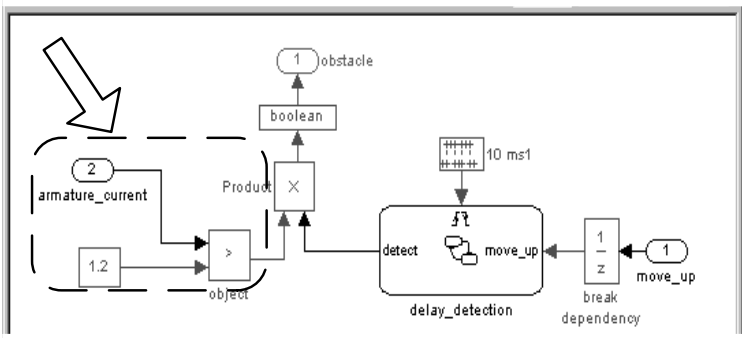


Figure 6: The position of the current comparison that is used to detect jammed windows[1].

After checking all indicators to the extent possible and documenting the issues as described, the defect detection phase is finished. Finally, in order to judge if the overall goal "Jam detection: Sensor data correctly received" is fulfilled, one has to go over the indicators and logical connectors once again. The left sub-goal "Dedicated jam detection" is not fulfilled. Regarding the right sub-goal "Jam detection by motor monitoring", each indicator is fulfilled, which results in fulfillment of the sub-goal and moreover to overall fulfillment of the goal, because the two sub-goals are connected via an "or"-node (meaning that only one of the sub-trees has to be fulfilled).

[1] Path inside the model:
powerwindow03/powerwindow_control_system/detect_obstacle_endstop/detect_obstacle

4 Conclusion and Future Work

This paper presented a novel approach that defines how an inspection could be performed using indicators in order to ensure certain quality goals. Output from existing analysis techniques like FMECA or FTA is used to identify risks and subsequently to derive goals and sub-goals, which are refined into indicators on certain artifacts, like requirements or models that can be checked by an inspector. We explained the approach with respect to a jam detection system where safety requirements should be ensured. With this, we could show that the defined approach is applicable to ensure the quality property safety (beside the initial focus on security) which gives initial evidence that the approach tends to be generalizable.

Quality assurance techniques like FMECA, FTA, or existing reading support for inspections often analyze corresponding documents on a level that is too coarse-grained, only identifying general risks. Unfortunately, only little or even no information is given on how to reduce the risk concretely. Thus, indicators are a new possibility to close this gap and to improve the artifacts to be checked with respect to the goals to be ensured, respectively the risk to be reduced. Consequently, the quality of the whole product is improved with a special focus on the qualities that are checked.

The indicator-based inspection is currently being evaluated on examples in the ViERforES research project [21]. In particular, a robot control system and an example from industrial automation are in the process of being examined using the techniques described in this paper, with more case studies planned later. The goal of these studies is to evaluate whether the techniques presented herein lead to an effective method for driving safety and security inspections (i.e., evaluation of the defect detection ability and effectiveness regarding certain quality properties, evaluation of scalability, evaluation of ensuring additional quality properties, and evaluation of the GIT construction step).

Another aspect regarding future work is an application of this approach to other domains, including relevant standards, and to further quality properties. Until now, we have been able to show that the indicator-based inspection approach is adaptable to the quality properties security [19] and safety (in this paper). We believe that more qualities can be refined with our approach and used for quality assurance in order to reduce risks in software development and thus, improve the products of practitioners.

Acknowledgements

This work was funded by the German Federal Ministry of Education and Research (BMBF) in the context of the project ViERforES (No.: 01 IM08003 B).

5 References

[1] M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal, 1976

[2] R. Ford, A. Howard, A Process for Performing Security Code Reviews, IEEE Security & Privacy, 2006
[3] J.C. Knight, E.A. Myers, An improved inspection technique, Communication of the ACM, 1993
[4] J. Martin, W.T. Tsai, N-fold inspection: a requirements analysis technique, Communication of the ACM, 1990
[5] O. Laitenberger, C. Atkinson, M. Schlich, K. El Eman, An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents. The Journal of Systems and Software, 53, p. 183-204, 2000
[6] C. Denger, M. Ciolkowski, F. Lanubile, Does active guidance improve software inspections? A preliminary empirical study, Proceedings of the IASTED International Conference Software Engineering, 2004
[7] F. Elberzhager, A. Klaus, M. Jawurek, Software Inspections using Guided Checklists to Ensure Security Goals, Secure Software Engineering Workshop, part of the ARES conference, 2009
[8] B. Brykczynski, A Survey of Software Inspection Checklists, Software Engineering Notes, vol. 24, no.1, ACM SIGSOFT, 1999
[9] A. Porter, L.G. Votta, Comparing Detection Methods for Software Requirements Specification: A Replication Using Professional Subjects, Empirical Software Engineering 3, p. 355-379, 1998
[10] The MathWorks: Matlab/Simulink Demo "Automative Power Window System", distributed with Matlab 2008a. A newer version of this example is available online at http://www.mathworks.com/products/simulink/demos.html?file=/products/demos/simulink/PowerWindow/html/PowerWindow1.html#3.
[11] ISO TS 22/SC3: ISO/DIS 26262: Road Vehicles – Functional Safety. Draft International Standard, ISO, 2009.
[12] A. Avižienis, J.-C. Laprie, B. Randell, C. Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Trans. on Dependable and Secure Computing, vol. 1, no. 1, 2004
[13] D. J. Lawson: Failure Mode, Effect and Criticality Analysis, Electronic System Effectiveness and Life Cycle Costing, J.K. Skwirzynski, ed., NATO ASI Series, F3, Springer-Verlag, Heidelberg, Germany, 1983, pp. 55–74
[14] Deutsche Gesellschaft für Qualität e. V. (DGQ): FMEA. Fehlermöglichkeits- und Einflussanalyse. Beuth-Verlag, Berlin, 2001.
[15] P.L. Goddard, Software FMEA techniques, Reliability and Maintainability Symposium, 2000. Proceedings. Annual , p. 118-123, 2000
[16] D.F. Haasl, Advanced Concepts in Fault Tree Analysis, presented at System Safety Symposium, 1965. Available at http://www.fault-tree.net/papers/haasl-advanced-concepts-in-fta.pdf.
[17] A. Birolini, Reliability Engineering: Theory and Practice, Springer, 2005.
[18] B. Schneier, Attack Trees: A formal, methodical way of describing the security of systems, based on varying attacks, Doctor Dobb's Journal, vol.24, p. 21–31, M&T publishing, 1999.
[19] H. Peine, M. Jawurek, S. Mandel, Security Goal Indicator Trees: A Model of Software Features that Supports Efficient Security Inspection, 11th IEEE High Assurance Systems Engineering Symposium, 2008
[20] IEC/DIN EN 61508, Funktionale Sicherheit sicherheitsbezogener elektrischer / elektronischer / programmierbar elektronischer Systeme, 2001
[21] www.vierfores.de, last visited: 2009-10-18
[22] Fault Tree Handbook, Systems and Reliability Research, Office of Nuclear Regulartory Research, U.S. Nuclear Regulartory Commission, Washington D.C., January 1981

Pseudo-Modelldifferenzen und die Phasenabhängigkeit von Metamodellen

Udo Kelter
Fachbereich Elektrotechnik und Informatik
Universität Siegen
kelter@informatik.uni-siegen.de

Abstract: Beim Vergleichen von Dokumenten werden manchmal Unterschiede angezeigt, die man als inhaltlich belanglos ansieht; solche Differenzen werden als Pseudodifferenzen bezeichnet. Wir betrachten dieses Phänomen für den speziellen Fall des Vergleichs von Modellen, deren Struktur durch ein Metamodell definiert wird, wie z.B. in der UML. Einen großen Teil der Pseudodifferenzen kann man darauf zurückführen, daß Metamodelle selbst abhängig von Entwicklungsphasen auf der Metaebene sind. Die Pseudodifferenzen entstehen hier, weil "spätphasige" Metamodelle benutzt werden. Weitere Typen von Pseudodifferenzen entstehen infolge von Editierkommandos bzw. elementaren Änderungen in abstrakten Syntaxgraphen, die nur durch mehrere zusammenhängende Änderungen auf der nächsttieferen Ebene realisiert werden können, ferner infolge suboptimaler Differenzen.

1 Einleitung

Beim Vergleichen von Dokumenten werden manchmal Unterschiede angezeigt, die man als inhaltlich belanglos ansieht; solche Differenzen werden als **Pseudodifferenzen** bezeichnet. Ein sehr einfaches Beispiel sind Leerzeichen am Ende von Textzeilen in einem Textdokument: man sieht sie nicht, egal wieviele vorhanden sind. Ein etwas komplexeres Beispiel mit Texten besteht darin, die Zahl der Leerzeichen zwischen zwei Worten belanglos zu finden, oder allgemeiner jeden beliebig geformten Leerraum zwischen zwei Worten als gleichwertig zu betrachten. Ein drittes Beispiel ist eine XML-Datei, in der zwei Realweltobjekte und eine Beziehung zwischen diesen Objekten repräsentiert werden, und zwar jedes Objekt durch ein XML-Element und die Beziehung durch zwei gleiche Werte in je einem Attribut in diesen beiden Elementen, z.B einem `ID`- und einem `IDREF`-Attribut. Welcher konkrete Wert in diesen beiden Attributen steht, ist belanglos, die dargestellte Beziehung bleibt die gleiche. Zwei XML-Dateien, die die gleichen Entitäten und Beziehungen repräsentieren, können daher umfangreiche textuelle Differenzen aufweisen.

Pseudodifferenzen müssen von echten Differenzen unterschieden werden, u.a. weil sie in Differenzdarstellungen stören und man dort nur echte Differenzen sehen will; bei Mischungen erzeugen sie unnötige Konflikte.

Die vorstehenden Beispiele legen es nahe, Äquivalenzen zwischen Dokumentzuständen

zu definieren und von einer Pseudodifferenz zu reden, wenn die Zustände äquivalent sind. Nur in einfacheren Fällen kann man Äquivalenzen lokal definieren (z.B. ein Leerzeichen ist äquivalent zu n Leerzeichen). Bei komplizierteren Dokumentstrukturen, z.B. der o.g. Beziehung in einer XML-Datei, liegt es nahe, von der textuellen Darstellung auf einen abstrakten Syntaxbaum überzugehen. Allerdings hilft dies in diesem Beispiel auch nicht weiter, denn die beiden Attributwerte, die die Beziehung darstellen, erscheinen auch im Syntaxbaum. Die offensichtliche Lösung besteht darin, zu einer noch abstrakteren Darstellung des Inhalts der XML-Datei überzugehen, die die Beziehung direkt enthält, hier also zu einem abstrakten Syntaxgraphen mit typisierten Knoten und Kanten.

In diesem Papier betrachten wir das Problem der Pseudodifferenzen speziell für Differenzen zwischen Modellen, deren Struktur wie z.B. in der UML durch ein Metamodell definiert ist. Eine Hauptthese ist, daß viele Pseudodifferenzen dadurch entstehen, daß "spätphasige" Metamodelle benutzt werden. Dies sind Metamodelle, die auf der Meta-Ebene technologiespezifische Anteile enthalten. Diese Formen von Pseudodifferenzen können nur systematisch behandelt werden, wenn man sich die Phasenabhängigkeit von Metamodellen explizit bewußt macht - letzteres geschieht in der Literatur bisher kaum [1].

Ferner werden teilweise die Phasenabhängigkeit und Meta-Ebenen miteinander verwechselt. Daher behandelt zunächst Abschnitt 2 die Phasenabhängigkeit von Metamodellen ausführlich und zeigt, daß Phasenabhängigkeiten unabhängig von linguistischen und ontologischen Metamodellhierarchien sind. Sobald man Metamodelle eindeutig Phasen zuordnet, kann man die meisten Pseudodifferenzen sehr einfach eliminieren, indem man auf "frühphasige" Metamodelle übergeht (s. Abschnitt 3).

Weitere Typen von Pseudodifferenzen entstehen infolge von elementaren Änderungen in abstrakten Syntaxgraphen, z.B. dem Löschen einer Kante, die nur durch mehrere zusammenhängende Änderungen in den technologieabhängigen Modellen realisiert werden können (s. Abschnitt 4). Ein analoger Effekt entsteht eine Größenstufe darüber durch elementare Änderungen in abstrakten Syntaxgraphen, die nicht isoliert durchgeführt werden können, sondern nur als Teil einer inhaltlich sinnvollen Editieroperation.

Es verbleiben einige unschärfer definierte Formen von Pseudodifferenzen, die abhängig von der Methode, wie Differenzen gewonnen werden, auftreten. Hier sind zustandsbasierte und protokollbasierte Verfahren zu unterscheiden, weil sie zunächst eigene Formen von überflüssigen Bestandteilen von Differenzen oder ungünstigen Darstellungen erzeugen können. In Abschnitt 5 zeigen wir, daß das Problem in beiden Fällen im Kern auf ein allgemeineres Optimierungsproblem hinausläuft, nämlich unter mehreren denkbaren Darstellungen der Unterschiede zwischen zwei Modellen eine günstige zu wählen.

[1] In einigen vielzitierten Publikationen über Metamodelle, z.B. [1, 2, 7, 10], wird die Phasenabhängigkeit von Metamodellen nicht diskutiert. Ob die Problematik übersehen oder bewußt ausgeklammert wurde, weil es primär um konzeptuelle Meta-Ebenen geht, sei dahingestellt. In [3], wo in Abschnitt 3.2 'From contemplative to operational models' auch praktische Fragen adressiert werden, werden Konversionen zwischen verschiedenen technologiespezifischen Repräsentationen von Metamodellen einfach als "Projektionen" bezeichnet; wie solche Projektionen arbeiten, bleibt offen. Publikationen zur Differenzberechnung von Modellen, z.B. diverse Beiträge zu den CVSM-Workshops [4, 5], gehen durchweg von vorgegebenen Metamodellen aus, die nicht weiter hinterfragt werden. Besonders gilt dies für "generische" Verfahren, die nur eine bestimmte Repräsentation der Metamodelle voraussetzen, z.B. als Ecore-Laufzeitobjekte.

2 Phasenabhängigkeit von Metamodellen

2.1 Paradigmen für Metamodellhierarchien

Metamodellhierarchien werden vor allem anhand des linguistischen und des ontologischen Paradigmas gebildet. Das linguistische Paradigma liegt der UML-Metamodellhierarchie [12] zugrunde, ebenfalls der wesentlich älteren CDIF-Metamodellhierarchie [6, 8]. Ontologische Metaebenen sind im Kern völlig unabhängig von den linguistischen [1, 2] und können, wenn man von linguistischen Ebenen ausgeht, auf jeder Ebene unabhängig voneinander entstehen. Bei beiden Paradigmen kann man die Entitäten der unteren Ebene als Instanzen von Entitäten der nächsthöheren Ebene auffassen, aber die Bedeutung der "ist-Instanz-von"-Beziehungen ist grundsätzlich anders.

In diesem Abschnitt fassen wir die wesentlichen Charakteristika dieser Hierarchien zusammen; auf dieser Basis können wir im folgenden Abschnitt klären, ob und wie Phasenabhängigkeiten von Metamodellen mit diesen Paradigmen korrelieren.

Merkmale linguistischer semantischer Ebenen. Die Metamodellebenen der UML basieren auf dem linguistischen Paradigma. Dieses ist analog zu den Metasprachebenen der Linguistik (Objektsprache, Metasprache, Meta-Metasprache, ...) definiert:

- Eine Aussage der Metasprache betrifft die Objekt*sprache als ganze* (Syntax, Grammatik, Semantik usw.), sie betrifft nicht einzelne Aussagen in der Objektsprache. Gegenstandsbereich der Metasprache ist die Objekt*sprache*, Gegenstandsbereich der Objektsprache ist die reale Welt. Beide Gegenstandsbereiche sind verschieden.
- Analog dazu macht ein Metamodell Aussagen bzw. repräsentiert Wissen über alle Modelle des zugehörigen Typs, namentlich wie die Modelle strukturiert und zu interpretieren sind. Metamodelle sind keine vereinfachten oder gekürzten Varianten der von Modellen. Die Gegenstandsbereiche beider Ebenen sind verschieden.

In beiden Fällen enthält die Metaebene also immer generelle Aussagen (oder Wissen oder Informationen) darüber, in welcher Form Aussagen (oder Wissen oder Informationen) der nächsttieferen Ebene sprachlich oder datenmäßig *repräsentiert* werden, also über die **Repräsentationsform** der nächsttieferen Ebene.

Merkmale ontologischer semantischer Ebenen. Eine ontologische Begriffshierarchie basiert auf einer Grundmenge von Entitäten, z.B. der Menge aller Tiere, und klassifiziert diese Entitäten anhand mehr oder minder abstrakter Begriffe, z.B. Hund - Raubtier - Säugetier - Wirbeltier. Ein abstrakterer Begriff gibt für seine Unterbegriffe und die zugehörigen Entitäten gemeinsame Merkmale und ggf. auch Merkmalsausprägungen vor. Auf Entitäten einer untergeordneten Gruppe treffen daher alle Merkmale und Merkmalsausprägungen aller übergeordneten Gruppen zu[2]. Die Gegenstandsbereiche aller Klassifikationsstufen sind *gleich*; im obigen Beispiel handelt es sich um auf allen Ebenen um

[2] Bei der Modellierung dieser Daten ist neben Typhierarchien oft das Typ-Instanz-Muster sinnvoll verwendbar.

Mengen von Lebewesen, die Teilmengen voneinander sind; wenn man die komplette Begriffshierarchie betrachtet, liefern die Ebenen dieses Baums jeweils andere Zerlegungen der Gesamtmenge aller hier betrachteten Lebewesen.

Wenn nun eine Begriffshierarchie so gestaltet ist, daß alle Wege von der Wurzel zu den Blättern gleich lang sind, kann man *durchgängige Ebenen* bilden und jeder Ebene einen Namen geben. Ein Beispiel sind die biologischen Klassifikationsstufen

Familie - Ordnung - Klasse - Unterstamm - Stamm.

Beispielsweise ist die Gruppe der Hunde eine Familie, und die Gruppe der Wirbeltiere *ist ein* Unterstamm. Man kann also von einer "is-a"-Beziehung zwischen "Wirbeltiere" und Unterstamm reden. Begriffe wie Stamm oder Ordnung klassifizieren keine einzelnen Lebewesen mehr, sondern Begriffe, insofern stehen sie auf einer höheren semantischen Ebene und stellen Meta-Begriffe dar. Für eine gegebene Menge von Entitäten kann es mehrere Klassifikationsmethoden geben, die zu unterschiedlichen Begriffshierarchien mit verschieden vielen Ebenen führen. Ein System von Klassifikationsstufen ist daher sehr eng verbunden mit einer konkreten Begriffshierarchie.

Bei Begriffshierarchien mit stark schwankenden Pfadlängen kann man i.d.R. keine sinnvollen Ebenen bilden und daher auch keine Klassifikationsstufen definieren (außer daß man die Ebenen von der Wurzel aus einfach durchnumeriert). In einem solchen Fall kann man also keine höhere ontologische semantische Ebene mehr bilden (während man in linguistischen Hierarchien prinzipiell immer höhere Ebenen bilden kann).

2.2 Modelle in den Entwicklungsphasen

Modelle treten in verschiedenen Entwicklungsphasen bzw. Verfeinerungsstufen eines Systems auf. Wenn Modell M2 die Weiterentwicklung von M1 ist, kann man häufig M2 als Ergebnis einer Transformation *trf* von M1 verstehen, zu der die weiteren Modellteile *Addendum* hinzugefügt wurden, also in einer mathematischen Schreibweise:

$$M2 = trf(M1) \cup Addendum$$

Ausgangspunkt sollten Modelle sein, die frei von technologiespezifischen Details sind (*platform independent model, PIM* [11]). Diese können in einem oder mehreren Schritten in technologiespezifische Modelle (*platform specific model; PSM*) und letztlich in Quellcode bzw. andere laufzeitrelevante Dokumente transformiert werden. Die Details dieser Transformationsketten hängen vom Typ der Modelle ab. Praxisrelevant sind solche Transformationen bei Datenmodellen, Zustandsmodellen und Ablaufstrukturmodellen.

Datenmodelle. Technologiefreie Modelle der Nutzdaten einer Applikation sind z.B. ER-Diagramme oder Analyse-Klassendiagramme, die wir i.f. als Analysedatenmodelle bezeichnen. Ein Analysedatenmodell wird in einem oder mehreren Schritten weiterentwickelt zu Datentypdefinitionen in einer konkreten Programmiersprache oder zu einem Schema für eine Datenbank oder für XML-Dateien (s. Bild 1), die für eine transiente bzw. persistente Darstellung der Nutzdaten benötigt werden. Die Typdefinitionen in Programmen bzw. Schemata in Datenverwaltungssystemen sind die präzisesten und detailliertesten Ent-

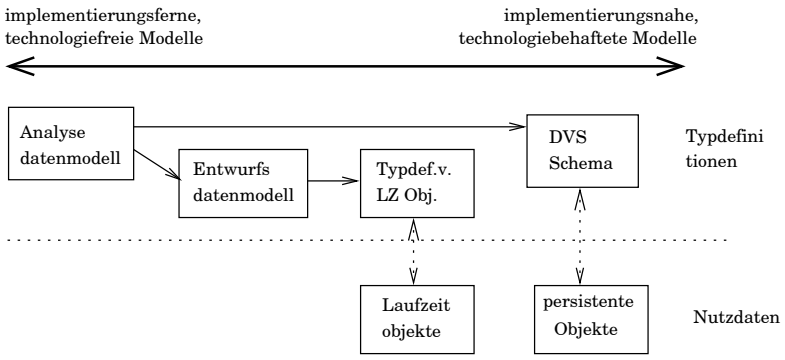


Abbildung 1: Modelle und Typdefinitionen eines Systems in verschiedenen Phasen

wicklungsdokumente; sie können nach einer Übersetzung bzw. Installation von einem Laufzeitsystem, das transiente oder persistente Instanzen verwalten kann, instantiiert werden.

Während die Schemata für die persistente Datenhaltung i.d.R. in einem einzigen Schritt aus den Analysedatenmodellen abgeleitet werden, sind für die transiente Seite mehrere Schritte üblich. Ein typisches Zwischenprodukt ist ein Entwurfsdatenmodell, das gegenüber dem Analysedatenmodell zusätzliche Modellelemente enthält, z.B. Containerklassen, und in dem die Navigationsrichtungen der Beziehungstypen festgelegt sind. Die Festlegung von Navigationsrichtungen ist ein Beispiel für eine technologiespezifische Entwurfsaufgabe bei transienten Daten, die für persistente Daten nicht existiert und die zu unterschiedlichen Transformationsketten führt.

Zustands- und Ablaufmodelle. Technologiefreie Zustandsmodelle von Systemen sind Zustandsübergangsdiagramme, Petri-Netze, *state machines* der UML und viele weitere Varianten. Technologiefreie Ablaufmodelle sind z.B. Aktivitätsdiagramme der UML. Endprodukte von Transformationsketten sind hier Teile des Quellcodes, die das Verhalten des Systems mitbestimmen, oder entsprechende Ressourcen (z.B. Zustandsübergangstabellen), die interpretiert werden. Die Transformationsketten müssen an die jeweilige Architektur des Zielsystems und die dort verwendeten Technologien angepaßt werden. Im Prinzip ergibt sich die gleiche Struktur wie in Bild 1), also mehrere von den technologiefreien Modellen ausgehende Transformationsketten.

“Modelle von Modellen”. Im Zusammenhang mit Pseudodifferenzen stellt sich die Frage, ob bei den Verfeinerungsstufen von Modellen auch eine Metamodellhierarchie vorliegt und ob und wie sie mit linguistischen bzw. ontologischen Hierarchien zusammenhängt.

Üblicherweise definiert man ein **Modell** eines (komplizierten, teuren, noch nicht vorhandenen, ...) Systems S als ein einfacheres System M, das interessierende Merkmale von S wiedergibt. Gemäß dieser Definition ist

– ein Analysedatenmodell ein Modell eines Entwurfsdatenmodells,
– ein Entwurfsdatenmodell ein Modell der resultierenden Typdefinitionen in einer Programmiersprache,
– der Quellcode ein Modell des bei der Übersetzung entstehenden Maschinen- oder Bytecodes bzw. des letztlich entstehenden lauffähigen Systems.

Wenn wir unter einem Metamodell generell das "Modell eines Modells" verstehen würden, dann folgte aus den vorstehenden Aussagen:

– ein Analysedatenmodell ist ein Modell eines Modells der Typdefinitionen, somit also ein *Metamodell* der Typdefinitionen.
– Wenn wir das Entwurfsdatenmodell in unserem Entwicklungsprozeß weglassen würden, wäre das Analysedatenmodell nur noch ein *Modell* der Typdefinitionen.
– Wenn wir den Entwurfsvorgang in zwei Schritte aufteilen, wird unser Entwurfsdatenmodell zu einem *Meta-Metamodell* der Typdefinitionen.

Die vorstehenden Beispiele zeigen, daß man *schrittweise Verfeinerungen bzw. Entwicklungsstufen eines Systems nicht sinnvoll als Metamodell-Ebenen auffassen kann*. Die Zahl der Entwicklungsstufen bzw. Transformationsschritte hängt nur vom individuellen Entwicklungsprozeß ab und ist insofern völlig willkürlich. Die Zahl der Entwicklungsstufen ist nicht an den Modellen selber erkennbar. Ferner ist die Zahl der Entwicklungsstufen für unterschiedliche Zieldokumente, z.B. Typdefinitionen der transienten bzw. persistenten Darstellungen derselben Daten, unterschiedlich.

Alle Modelle, die schrittweise Entwicklungsstufen eines Systems sind, modellieren das gleiche Endprodukt und haben daher *den gleichen Gegenstandsbereich*! Entwicklungsstufen von Modellen sind daher völlig orthogonal zu linguistischen Metaebenen, denn linguistische Metaebenen haben unterschiedliche Gegenstandsbereiche.

Entwicklungsstufen korrelieren auch nicht mit ontologischen Ebenen. Ausgangspunkt ontologischer Metaebenen ist immer eine Gesamtmenge von zu klassifizierenden Entitäten, die alle konzeptuell auf dem gleichen Niveau nebeneinander stehen. Modelle des gleichen Systems auf verschiedenen Entwicklungsstufen stehen in diesem Sinne nicht unabhängig nebeneinander, sondern überlappen inhaltlich erheblich. Klassifiziert wird allenfalls die Gesamtmenge aller denkbaren Realisierungen des Systems, das durch die gegebenen Analysemodelle spezifiziert ist; der Klassifikationsbaum würde dann aus allen Verfeinerungen bestehen, die auf der jeweils nächsten Entwicklungsstufe denkbar sind. Diese Gesamtmenge interessiert aber gar nicht, ihre Struktur ist nicht Modellierungsgegenstand.

2.3 Phasenabhängigkeit der Metamodelle eines Modelltyps

Aus den vorstehenden Betrachtungen ergibt sich unmittelbar, daß alle Modelltypen, die in Bild 1 auf der Ebene "Typdefinitionen" angeordnet sind, eigene Repräsentationsformen benötigen, also insb. dann, wenn man die Modelle selber maschinell verarbeiten muß, *eigene (linguistische) Metamodelle* benötigen!

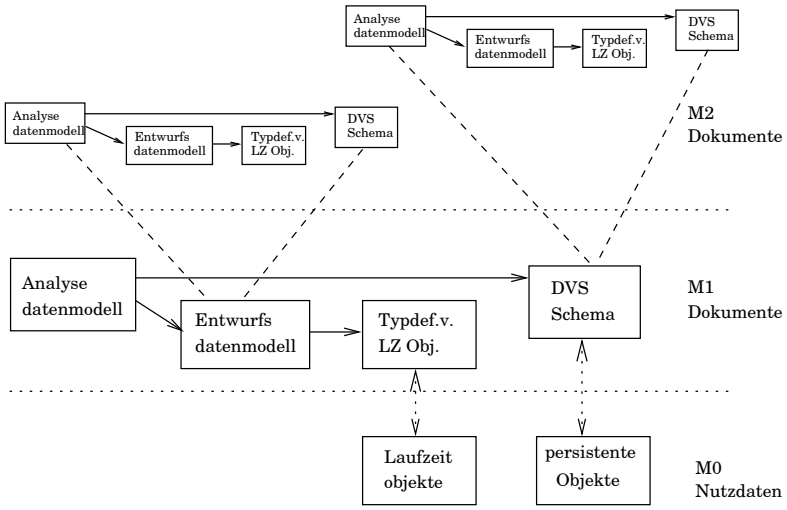


Abbildung 2: Phasenabhängige Metamodelle von Modellen

Wir betrachten als Beispiel eines Modells auf Ebene M1 der OMG-Modellhierarchie ein Entwurfs-Datenmodell, s. Bild 2. Man könnte glauben, mit einem einzigen (M2-) Metamodell für diesen (M1-) Modelltyp auszukommen. Dies trifft aber nicht zu: wir benötigen

1. *abstraktere* Metamodelle (also Analyseklassendiagramme), die man in den frühen Phasen der Entwicklung von Modellierungswerkzeugen einsetzen wird; derartige Metamodelle werden z.B. in der Spezifikation der UML [13] eingesetzt;
2. *implementierungsnahe* Metamodelle für *transiente* Repräsentationen von Modellen als Laufzeitobjekte in einer bestimmten Programmiersprache, z.B. in Modelleditoren;
3. *implementierungsnahe* Metamodelle für die *persistente* Repräsentationen von Modellen in Dateien oder Datenbanken (üblicherweise als Schema bezeichnet), z.B. zum Dokumentaustausch oder zur Archivierung.

Im Prinzip liegen die gleichen Verhältnisse wie in Bild 1 gezeigt vor, nur ist dort der Begriff "Nutzdaten" der Ebene M0 zu ersetzen durch "Repräsentationen von Modellen", also die Nutzdaten von Modelleditoren. Modelle sind, wenn sie z.B. in MDD-Ansätzen maschinell verarbeitet werden, völlig normale Daten, zu deren Modellierung und Implementierung die üblichen Entwicklungsschritte zu durchlaufen sind. Diese Erkenntnis ist eigentlich trivial, wird in der Literatur aber weitgehend übersehen.

Übersehen wird ferner meist auch, daß Daten zusammen mit den Operationen auf den Daten entwickelt werden sollten; Operationen auf Modellen sind vor allem die Editierkommandos von Modelleditoren; auf diese gehen wir anschließend noch näher ein.

3 Pseudodifferenzen infolge implementierungsnaher Metamodelle

In der Einleitung genannt war als ein Beispiel für Pseudodifferenzen ein Paar von XML-Dateien, die die gleichen Entitäten und Beziehungen repräsentieren – die Datei könnte ein Modell repräsentieren – und die trotzdem umfangreiche textuelle Differenzen aufweisen. Der Begriff Pseudodifferenz unterstellt dabei das in Bild 3 gezeigte Szenario: Es liegen zwei technologiespezifische Repräsentationen (spätphasige Modelle) R1 und R2 vor, die das "gleiche" abstrakte Modell repräsentieren und die trotzdem Unterschiede aufweisen. Allgemeiner ist das Szenario auf separat darstellbare, identische Teile von zwei verschiedenen abstrakten Modellen zu beziehen. Eine der beiden Repräsentationen kann durch Anwendung der Transformationsfunktion *trf* entstanden sein, die andere z.B. durch Editiervorgänge, die im Endeffekt nichts verändert haben, z.B. Laden und unverändertes Abspeichern in einem Editor. R1 und R2 müssen keine Versionen voneinander sein, sondern können unabhängig entstanden sein.

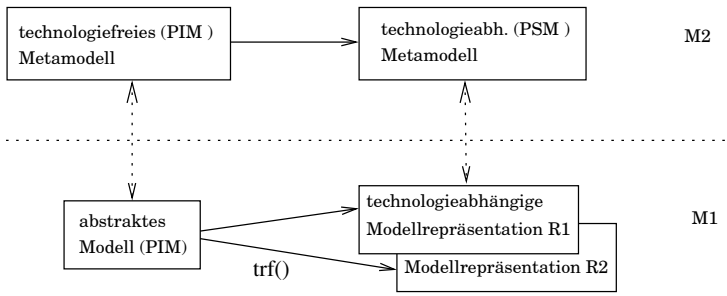


Abbildung 3: Szenario einer Pseudodifferenz

Wir unterstellen, daß die abstrakten Modelle als abstrakte Syntaxgraphen angesehen werden können. Die technologiespezifischen Implementierungen dieser abstrakten Syntaxgraphen enthalten Details, die konzeptuell nicht auftreten und die Pseudodifferenzen verursachen können. Details hierbei hängen stark davon ab, wie (un-) geschickt die Transformationsfunktion *trf* gestaltet ist und wie die jeweiligen Werkzeuge arbeiten. Häufig entstehen Pseudodifferenzen bei der Implementierung folgender konzeptueller Strukturen:

- Beziehungen (Kanten im abstrakten Syntaxgraph): Diese werden in persistenten Repräsentationen durch zwei gleiche Werte bzw. in transienten Repräsentationen oft durch zwei gegenläufige Referenzen repräsentiert. Die Datenwerte sind beliebig austauschbar, die Werte dieser Referenzen sind zufällig. Ein Austausch führt zu zwei lokalen Pseudodifferenzen.
- ungeordnete Kollektionen von Modellelementen, die im abstrakten Syntaxgraph durch ein Wurzelobjekt und von dort ausgehende Kanten eines bestimmten Typs bestimmt werden: In XML-Dateien sind Kollektionen durch die Dokumentreihenfolge geordnet, durch unterschiedliche Anordnungen können hier Pseudodifferenzen entstehen.
- Analog gilt für geordnete Kollektionen, daß die Ordnung in relationalen Tabellen oder anderen Strukturen, die per se ungeordnet sind, explizit implementiert werden muß,

z.B. durch laufende Nummern, und hier bei der Neuvergabe der Nummern Pseudodifferenzen entstehen können.

Vermeidung von Pseudodifferenzen. Pseudodifferenzen sind normalerweise unerwünscht. Es gibt zwei grundlegende Methoden, um sie zu vermeiden:

1. normierte Darstellungen in den implementierungsnahen Modellrepräsentationen,
2. Rekonstruktion der frühphasigen Modelle und Vergleich auf deren Basis.

Die erste Methode ist nur unter speziellen Randbedingungen, auf die hier nicht eingegangen werden soll, realisierbar. U.a. muß jedes abstrakte Modellelement einen universell eindeutigen Identifizierer haben, um eindeutige Darstellungen von Beziehungen zu ermöglichen. Ferner müssen irrelevante und relevante Teile der Modelle unterscheidbar sein, was bei der Gestaltung der spätphasigen Metamodelle beachtet werden muß. Verglichen werden die spätphasigen Modelle, die Vergleichsfunktion wird aber dahingehend modifiziert, Unterschiede in den irrelevanten Teilen auszublenden.

Die zweite Methode kann bei transienten Repräsentationen von Modellen oft durch passende Interfaces oder Adapter zu den ohnehin vorhandenen Modellrepräsentationen realisiert werden, die überflüssige Teile ausblenden. Das Kopieren des Modells kann dann vermieden werden. Die zweite Methode empfiehlt sich besonders für persistente Repräsentationen von Modellen in XML-Dateien. Standardverfahren zum Vergleich von Texten versagen hier weitgehend, wenn spezielle Vergleichsverfahren implementiert werden, müssen die Dateien ohnehin in transiente Darstellungen eingelesen werden und können dabei konvertiert werden.

In vielen Modelleditoren kann man den "Konstruktionsfehler" beobachten, daß einfach die komplette transiente Modellrepräsentation mit Standardverfahren in eine XML- (bzw. XMI-) Darstellung konvertiert wird. Das Resultat enthält dann nicht nur die XML-spezifische Besonderheiten, sondern auch noch Zufälligkeiten aus der transienten Repräsentation und schlimmstenfalls noch Hilfsdaten für interne Editorfunktionen.

4 Nichtatomare Implementierungen konzeptueller Editieroperationen

Die Metamodelle der UML und anderer Modellierungssprachen sind reine Datenmodelle. Implizit unterstellt wird, daß die (M1-) Modelle als abstrakte Syntaxgraphen repräsentiert werden. Die Metamodelle beschreiben nur noch die Typen der Knoten, Kanten, Attribute und weitere Details dieser Graphen.

Modelle modellieren jedoch eigentlich nicht nur Daten, sondern Systeme, die Funktionen anbieten. Als Funktionen implizit vorausgesetzt werden die elementaren Graphoperationen, namentlich das Erzeugen und Löschen von Knoten und Kanten oder das Setzen von Attributen.

Bei den meisten Modelltypen und Typen von Modellelementen der UML ist jede zulässige elementare Operation im abstrakten Syntaxgraphen, sofern sie im Rahmen von OCL-

Constraints überhaupt zulässig ist, ein sinnvoller Editierschritt. Beim Vergleich zweier abstrakter Syntaxgraphen, in denen ein Modellelement vorhanden ist bzw. fehlt, ist der Rückschluß möglich, daß es erzeugt bzw. gelöscht worden ist.

Nichtatomare Implementierungen elementarer Operationen im abstrakten Syntaxgraphen. Der vorgenannte einfache Rückschluß ist bei spätphasigen Modellen leider nicht immer möglich. Ein Beispiel ist das Erzeugen oder Löschen einer Kante im abstrakten Syntaxgraphen: bei vielen Implementierungen der abstrakten Syntaxgraphen führt dies zu zwei lokalen Unterschieden in den spätphasigen Modellen. Je nach Modelltyp und Implementierung der Kanten können sich auch mehr als zwei Unterschiede in den spätphasigen Modellen ergeben. Wenn man nun einen dieser lokalen Unterschiede als echt, also Repräsentanten der konzeptuellen Änderung ansieht, sind alle anderen "unecht". Bei mehreren zusammenhängenden Änderungen im abstrakten Syntaxgraphen können komplizierte Konstellationen "unechter" Änderungen in den spätphasigen Modellen entstehen.

Nichtatomare Operationen im abstrakten Syntaxgraphen. Darüber hinaus sind manche elementaren Operationen im abstrakten Syntaxgraphen für sich alleine nicht zulässig bzw. sinnvoll. Ein Beispiel sind Assoziationen in Klassendiagrammen. Im abstrakten Syntaxgraphen wird eine Assoziation durch einen Knoten für die Assoziation und wenigstens zwei Knoten für die Assoziationsenden und eventuell weitere Knoten repräsentiert; hinzu kommen diverse Kanten. Aus Benutzersicht sind einzelne Änderungen an diesen Knoten und Kanten nicht sinnvoll[3], sinnvolle Editieroperationen sind beispielsweise

- das Anlegen bzw. Löschen einer ganzen Assoziation, wofür ein ganzer Teilbaum im abstrakten Syntaxgraphen angelegt bzw. gelöscht werden muß
- das Ändern der Klasse, die eine Rolle einnimmt.

Die Menge der nichtatomaren Operationen ergibt sich in solchen Fällen *nicht automatisch aus den Datenstrukturen*, sondern muß in einer bewußten Entwurfsentscheidung festgelegt werden. Weitere Beispiele in Zustandsmodellen, bei denen elementare Operationen im abstrakten Syntaxgraphen nicht verwendet werden können, sind in [9] beschrieben.

Wenn man die Wurzel des Teilbaums, der eine Assoziation implementiert, als Repräsentanten der Assoziation ansieht und die Löschung dieses Knotens als Löschung der Assoziation, sind alle anderen dadurch implizierten Löschungen von Knoten und Kanten in diesem Teilbaum "unecht". Ähnliche Fälle treten bei allen konzeptuellen UML-Modellelementen auf, die durch mehrere Knoten und Kanten repräsentiert werden. OCL-Constraints, die ein isoliertes Ändern einzelner Knoten oder Kanten verbieten, verursachen ebenfalls implizierte Änderungen; Beispiele hierfür sind diverse *subsets*-Constraints zwischen Mengen von Kanten.

Man kann den Begriff Pseudodifferenz auf die vorgenannten "unechten" Änderungen ausdehnen. Hauptmerkmal der Situation ist, daß eine atomare Operation auf der Ebene von Benutzerkommandos oder im abstrakten Syntaxgraphen nur durch mehrere lokale Operationen auf der nächsttieferen Ebene (abstrakter Syntaxgraph bzw. spätphasiges Modell)

[3] Ferner führen sie zu inkonsistenten abstrakten Zuständen, die kein gültiges Modell mehr darstellen.

realisiert werden kann und nicht jede Änderung auf der unteren Ebene als Repräsentant einer konzeptuellen Änderungen gewertet werden kann.

5 Suboptimale Differenzen

Über die bisher diskutierten strukturellen Ursachen hinaus können "belanglose" bzw. uninteressante Anteile von Differenzen durch "suboptimale" Differenzen entstehen. Details hängen von der Methode ab, wie Differenzen bestimmt werden und welche "natürliche" Repräsentation von Differenzen sich daraus ergibt:

1. **Zustandsbasierte Verfahren** vergleichen die Zustände der beiden Modelle. Sie bestimmen zunächst "gleiche", also korrespondierende Modellelemente, die den Durchschnitt der Elementmengen darstellen. Im Prinzip liegt hier eine symmetrische Differenz vor, die als Tabelle von Korrespondenzen zwischen Modellelementen und Einfügungen in die gemeinsamen Teile darzustellen ist.
2. **Protokollbasierte Verfahren**: diese zeichnen Editierkommandos in Editoren auf.

Ein Optimierungsproblem bei zustandsbasierten Verfahren entsteht durch komplexe Editierkommandos. Ein Beispiel hierfür in einem Zustandsmodell ist das Verschieben eines Zustands in eine innere Region eines anderen Zustands. Diese Änderung kann auch (umständlich) durch Benutzerkommandos realisiert werden, die den alten Zustand löschen und ihn in der inneren Region neu anlegen. Bei einem Zustandsvergleich findet man zunächst genau diese elementaren Editierschritte. Aus Benutzersicht ist diese Differenzdarstellung aber sehr ungeschickt und mit uninteressanten Details belastet. Mit einer Verschiebung kann man die Änderung wesentlich besser darstellen.

Bei protokollbasierten Verfahren kann z.B. ein Modellelement zuerst erzeugt und später wieder gelöscht worden sein, d.h. auch hier können uninteressante Teile auf treten.

Bei beiden Methoden, Differenzen zu bestimmen, treten jeweils eigene Fälle auf, in denen die Differenz umständlich bzw. mit verzichtbaren Details belastet erscheint, also durch eine bessere Differenz ersetzt werden sollte; Details können hier aus Platzgründen nicht diskutiert werden. Solche suboptimalen Differenzen kann man allenfalls noch am Rande unter dem Begriff Pseudodifferenz subsumieren, sie sind letztlich Sonderfälle eines allgemeineren Optimierungsproblems und sie haben strukturell andere Ursachen als die vorher diskutierten Arten von Pseudodifferenzen.

6 Zusammenfassung

Wenn man Modelle vergleicht, können sie technisch nur in einer technologiespezifischen persistenten oder transienten Repräsentation vorliegen. Hierbei kommt es regelmäßig zu Pseudodifferenzen, also lokalen Unterschieden, die als irrelevant angesehen werden. Hauptziel dieses Papiers ist eine Klassifizierung der Ursachen für diese Unterschiede.

Die wichtigste Ursache für Pseudodifferenzen sind technologiespezifische Anteile in den Repräsentationen der Modelle. Die zugehörigen Metamodelle kann man als technologiebehaftet oder "spätphasig" bezeichnen, weil sie in den späten Entwicklungsphasen von Werkzeugen, die Modelle verarbeiten, benötigt werden. Für jeden M1-Modelltyp existieren daher mehrere technologiefreie bzw. -behaftete Metamodelle. Unterschiedliche M1-Modelltypen haben im Prinzip immer eigene Metamodelle.

Pseudodifferenzen zwischen Modellen stören, interessiert ist man nur am Vergleich des "konzeptuellen" Inhalts der Modelle. Letzteren kann man in den meisten Fällen als abstrakten Syntaxgraphen auffassen. Man muß also von den Unterschieden in den technologiespezifischen Repräsentationen auf konzeptuelle Unterschiede zurückschließen. Wenn die Metamodelle geschickt gewählt sind, findet man eindeutige Repräsentanten der Knoten, Kanten und Attribute des abstrakten Syntaxgraphen; von Änderungen an diesen Repräsentanten kann auf konzeptuelle Änderungen zurückgeschlossen werden.

In manchen Fällen stellen indes elementare Änderungen im abstrakten Syntaxgraphen keine sinnvollen Editieroperationen dar; hier ist eine weitergehende Abstraktion erforderlich, die über die reine Datenmodellierung hinausgeht, nämlich die Definition eines Editierdatentyps, der die zulässigen Operationen auf abstrakten Modellen festschreibt. Das Problem, geeignete Repräsentanten für die Durchführung von Editieroperationen zu finden, stellt sich hier erneut, aber eine Abstraktionsstufe höher.

Literatur

[1] Atkinson, Colin; Kühne, Thomas: Rearchitecting the UML Infrastructure; ACM Transactions on Modeling and Computer Simulation 12:4, p.290-321; 2002

[2] Atkinson, Colin; Kühne, Thomas: Model-Driven Development: A Metamodeling Foundation; IEEE Software 20:5, p.36-41; 2003

[3] Bézivin, Jean: On the Unification Power of Models; Software and Systems Modeling 4:2, p.171-188; 2005

[4] Ebert, Jürgen; Kelter, Udo; Systä, Tarja: Proc. 2008 Intl. Workshop on Comparison and Versioning of Software Models; ACM, ISBN 978-1-60558-045-6; 2008

[5] Ebert, Jürgen; Kelter, Udo; Systä, Tarja: Proc. 2009 ICSE Workshop on Comparison and Versioning of Software Models; IEEE, ISBN 978-1-4244-3714-6; 2009

[6] Flatscher, Rony G.: Metamodeling in EIA/CDIF—Meta-Metamodel and Metamodels; ACM ToMaCS 12:4, p.322-342; 2002

[7] Hesse, Wolfgang: More matters on (meta-)modelling: remarks on Thomas Kühne's "matters"; Journal on Software and Systems Modeling 5:4, p.387-394, December 2006; Springer; 2006

[8] Imber, Mike: The CASE Data Interchange Format (CDIF) standards; p.457-474 in: Software Engineering Environments; Ellis Horwood; 1991 ISBN 0-13-832601-0

[9] Kelter, Udo; Schmidt, Maik: Comparing State Machines; p.1-6 in [4]

[10] Kühne, Thomas: Matters of (Meta-) Modeling; Journal on Software and Systems Modeling 5:4, p.369-385, December 2006; Springer; 2006

[11] MDA Guide Version 1.0.1; OMG, Doc. omg/2003-06-01; 2003

[12] Meta Object Facility (MOF) Core Specification, Version 2.0; OMG, formal/06-01-01; 2006

[13] Unified Modeling Language: Superstructure, Version 2.0; OMG, Doc. formal/05-07-04; 2006

Objektrelationale Programmierung

Dilek Stadtler
Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
dilek.stadtler@fernuni-hagen.de

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
steimann@acm.org

Abstract: Bislang gelten vor allem objektrelationale Datenbanken als Antwort auf den sog. Impedance mismatch zwischen den Welten der relationalen Datenhaltung und der objektorientierten Programmierung. Angesichts jüngster Bestrebungen, im Gegenzug relationale Elemente in die objektorientierte Programmierung einzubringen (wie etwa mit Microsofts LINQ-Projekt), zeigen wir auf, wie das inhärent Zeiger dereferenzierende Modell der objektorientierten Programmierung erweitert werden kann, so daß sich auch relationale Teile eines Datenmodells direkt, d. h. ohne Ergänzung umfangreichen stereotypen Codes, in objektorientierte Programme umsetzen lassen.

1 Einleitung

Annähernd parallel zum Aufkommen der objektorientierten Programmierung fanden relationale Datenbanksysteme breiten Einzug in die kommerzielle Praxis. Da beide Entwicklungen für sich genommen ausgesprochen erfolgreich waren, findet man heute häufig Konstellationen vor, in denen objektorientierte Programme auf relationalen Datenbeständen operieren müssen. Unglücklicherweise unterscheiden sich die beiden Paradigmen gleich in mehreren Eigenschaften so sehr, daß an der Schnittstelle zwischen Datenhaltung und Programmierung ein beträchtlicher Übergangswiderstand[1] auftritt. Diesen gilt es zu beseitigen.

Nachdem dazu zunächst mit vergleichsweise wenig Erfolg versucht wurde, relationale Datenbanken durch objektorientierte abzulösen, findet man heute vielerorts sog. objektrelationale Datenbanksysteme vor, an die sich objektorientierte Programme anbinden können. Die Annäherung der Paradigmen erfolgt dabei allerdings recht einseitig von Seiten der Datenbanken. Tatsächlich werden Programmierer so zumindest weitestgehend von der Last befreit, die Modelle einer vielfältigen Anwendungsrealität auf die vergleichsweise starren Tabellen einer relationalen Datenbank herunterbrechen zu müssen. Auf der Strecke geblieben ist dabei jedoch die Relation als fundamentale konzeptuelle Abstraktion.

In jüngerer Zeit ist eine Reihe von Arbeiten entstanden, die den umgekehrten Weg gehen und versuchen, die Vorteile einer relationalen Sicht auf Daten in die objektorientierte

[1] In der englischsprachigen Literatur spricht man von einem „impedance mismatch“ [CM84].

Programmierung einzubringen, indem sie objektorientierte Programmiersprachen um relationale Konstrukte erweitern [BW05, NPN08, Øs07]. Die meisten dieser Arbeiten führen dazu Relationen als neben Klassen gleichberechtigte Sprachkonstrukte ein, deren Instanzen die herkömmliche Verzeigerung der Objekte über Instanzvariablen und Collections ersetzen sollen. Eine der sichtbarsten Arbeiten auf diesem Gebiet ist jedoch ausgerechnet Microsofts LINQ-Projekt, das gerade nicht die Möglichkeiten der Datenstrukturierung um Relationen ergänzt, sondern vielmehr eine (an SQL angelehnte) relationale Abfragesprache für die herkömmlichen, collection-basierten Datenstrukturen einführt [BMT07]. Die Datenstrukturen, die einem objektorientierten Programm zugrunde liegen, werden dadurch jedoch nicht relationaler.

Eine von beiden Entwicklungsrichtungen abkehrende und weder auf Seiten der Datenbanken noch auf Seiten der objektorientierten Programmierung anzusiedelnde Lösung stellt der Einsatz von objektrelationalen Mapping-Werkzeugen (O/R-Mapping) dar. Mit Werkzeugen dieser Art wird versucht eine Middleware-Lösung für das Problem zu finden, indem objektorientierte Programme innerhalb eines Persistenzlayers in relationale Datensätze umgewandelt und anschließend in einem relationalen Datenbanksystem gespeichert werden. Aus theoretischer Sicht kann dieser Ansatz jedoch nicht als Lösung zur Verringerung des Impedance mismatch betrachtet werden, da hier lediglich eine Brücke zwischen den zwei (immer noch sehr unterschiedlichen) Datenmodellierungskonzepten geschlagen, eine angemessene Annäherung beider Konzepte jedoch nicht in Betracht gezogen wird. Aus praktischer Sicht ist neben den teilweise relativ großen Performance-Problemen der Rückschritt in Bezug auf die vom Anwendungsprogrammierer abverlangten Kenntnisse der relationalen Speicherung zu nennen. (Denn trotz teilweise vorhandener automatisierter Mapping-Werkzeuge kommt der Programmierer insbesondere bei den gängigen Werkzeugen nicht ohne Kenntnisse des zugrundeliegenden Mappings aus.)

Mit dieser Arbeit wollen wir den Impedance mismatch verringern, indem wir die beiden zuerst genannten Stoßrichtungen vereinen und eine sanfte Erweiterung des objektorientierten Datenmodells vorstellen, die seinen navigierenden (d. h. im wesentlichen Zeiger dereferenzierenden) Charakter erhält, also insbesondere ohne die Einführung von Relationen als separat zu verwaltenden Tupelmengen auskommt. Ziel ist hierbei eine Annäherung des objektorientierten Programmiermodells an das relationale Modell, so dass nur die wesentlichen diesbezüglichen Defizite des objektorientierten Datenmodells beseitigt und ein aus Anwendersicht komfortabler Umgang mit dem (erweiterten) objektorientierten Programmiermodell möglich wird. Unser Vorhaben erstrebt somit die Erweiterung gängiger objektorientierter Programmiersprachen und bezieht die Alternative der Entwicklung gänzlich neuer (datenbankintegrierter) Systeme nicht ein. Als Ansatzpunkte hierfür haben wir in einer parallelen Arbeit [SS09] die implementationsbedingte Unterscheidung zwischen Zu-1-Beziehungen (direkt über Zeiger realisiert) und Zu-*n*-Beziehungen (per Umweg über Collections realisiert) sowie die mangelnde Bidirektionalität von Beziehungen, die bislang durch paarige Beziehungen kompensiert werden muß (in der Natur der Zeiger begründet), ausgemacht. Indem wir bidirektionale Beziehungen einführen und uni- und bidirektionale Zu-1- und Zu-*n*-Beziehungen so vereinheitlichen, daß sie sich syntaktisch nur noch bei ihrer Deklaration unterscheiden, wollen wir den Weg für eine Programmierung bereiten, die wir (in Anlehnung an die objektrelationalen Datenbanken, aber dazu im Ansatz eher komplementär) *objektrelational* nennen.

```
class Firma {
  ICollection<Angestellter> angestellte = new List<Angestellter>();
  ICollection<Arbeitsplatz> arbeitsplätze = new List<Arbeitsplatz>();

  ICollection<Arbeitsplatz> freieArbeitsplätzeMitTelefon() {
     ICollection<Arbeitsplatz> ergebnis = new List<Arbeitsplatz>();
     foreach (Arbeitsplatz aplz in arbeitsplätze)
        if (aplz.mitTelefon) ergebnis.Add(aplz);
     foreach (Angestellter agst in angestellte)
        if (agst.arbeitsplatz != null) ergebnis.Remove(agst.arbeitsplatz);
     return ergebnis;} }
class Angestellter { Arbeitsplatz arbeitsplatz; }
class Arbeitsplatz { bool mitTelefon; }
```

Abbildung 1: Beispielhafte unidirektionale Zu-1- und Zu-*n*-Beziehungen sowie eine darauf basierende Auswertung, die die Umkehrung einer Beziehung verlangt (in C#).

Der Rest der Arbeit gliedert sich wie folgt: In Abschnitt 2 stellen wir das Problem aus unserer Sicht vor und diskutieren kurz die Arbeiten, die sich damit bereits befaßt haben. In den Abschnitten 3 und 4 stellen wir dann unsere Lösungen für die beiden obengenannten Einzelprobleme vor und zeigen, wie sie sich in die konventionelle objektorientierte Programmierung eingliedern. In Abschnitt 5 skizzieren wir noch kurz die Implementierung (mittels Bibliotheken) in C#, bevor wir in Abschnitt 6 zusammenfassen und schließen.

2 Probleme

2.1 Das Problem der Unidirektionalität

Die Unidirektionalität, also die Tatsache, daß alle Beziehungen gerichtet und nur in diese Richtung navigierbar sind, ist immer dann ein Problem, wenn ein Sachverhalt Navigation in beide Richtungen erfordert. Hier zwei Beziehungen koordiniert pflegen zu müssen, stellt gegenüber dem Relationenmodell, in dem man es von Haus aus immer nur mit *einer* Beziehung zu tun hat, die in beliebige Richtungen navigierbar ist, einen erheblichen Nachteil dar. Ein Beispiel soll dies verdeutlichen.

Eine Firma verfüge über eine Reihe von Angestellten und eine Menge von Arbeitsplätzen, von denen manche mit einem Telefon ausgestattet sind. Einem Angestellten sei ein Arbeitsplatz zugeordnet — die umgekehrte Zuordnung sei jedoch nicht modelliert. Sie läßt sich aber aus den bestehenden Beziehungen ableiten, wie das Beispiel in Abbildung 1 zeigt. Die indirekte Form der Ableitung wird daran deutlich, daß die zweite Schleife in der Methode freieArbeitsplätzeMitTelefon() über die Angestellten und nicht über die Arbeitsplätze iteriert. Während dies im gegebenen Beispiel vielleicht kein großes Problem ist, so findet man in der Realität doch leicht Varianten, die nicht nur umständlich zu programmieren, sondern auch noch ausgesprochen ineffizient in der Ausführung sind.

Eine mögliche Lösung ist, die umgekehrte Richtung der Beziehung ebenfalls explizit vorzusehen. Das Problem hierbei ist allerdings, daß die Anpassung der Inhalte der beiden Instanzvariablen bei einer Zuweisung an eine der beiden für die andere ebenfalls explizit (per Zuweisung) erfolgen muß. Dies ist nicht nur lästig, sondern auch noch fehleranfällig.

Obwohl bidirektionale Beziehungen im relationalen Modell durch Schlüssel-Fremdschlüssel-Kombinationen leicht umzusetzen sind, besteht dieses Problem bei Verwendung von objektrelationalem Mapping im Kern in gleichem Maße, da bei gängigen O/R-Werkzeugen Bidirektionalität auf die Navigierbarkeit (die bei objektorientierten Programmiersprachen generell gegeben ist) und nicht zusätzlich auf die gegenseitige Synchronisierung, die das eigentliche Problem darstellt und die auch hier dem Implementierer überlassen bleibt, reduziert wird. Im Fall von vorhandenen automatisierten Synchronisierungskonzepten (die i.d.R. starke Performanceeinbußen mit sich bringen) kann diese Alternative dennoch nicht als zufriedenstellend im Sinne des objektorientierten Modells betrachtet werden, da bidirektionale Beziehungen dann ausschließlich in Kombination mit objektrelationalen Datenbanken möglich sind. Der Bedarf einer anderen, von objektrelationalen Datenbanken unabhängigen Lösung wird somit deutlich.

2.2 Das Problem der Unterscheidung von Zu-1- und Zu-*n*-Beziehungen

Durch die Verweissemantik von Variablen in der objektorientierten Programmierung stellt jede belegte Instanzvariable eine gerichtete Beziehung zu genau einem anderen Objekt her. Diese Beziehung zu navigieren entspricht der Dereferenzierung des Verweises (Zeigers) und ist damit extrem effizient umgesetzt. Wird der Variable ein neues Objekt zugewiesen, wird damit die Beziehung zum ursprünglichen Objekt durch die zum neuen ersetzt.

Da eine Variable nicht auf mehrere Objekte gleichzeitig verweisen kann, ist für die Umsetzung von Zu-*n*-Beziehungen der Umweg über Zwischenobjekte notwendig. Eine Instanzvariable, die ein solches Zwischenobjekt zum Wert hat, verweist damit aber nicht auf die im Zwischenobjekt enthaltenen Objekte, sondern auf das Zwischenobjekt. Eine Zuweisung an diese Variable ersetzt somit auch nicht (wie im Zu-1-Fall) das bezogene Objekt (selbst dann nicht, wenn es tatsächlich nur eines ist), sondern das Zwischenobjekt. Soll sich die Menge der bezogenen Objekte ändern, ist dazu das Zwischenobjekt zu manipulieren.

Dieser fundamentale Unterschied in der Umsetzung von Zu-1 und Zu-*n*-Beziehungen ist zwar implementierungstechnisch nachvollziehbar, aber logisch nicht gerechtfertigt. Schnell kann sich aus der Anwendungsdomäne heraus ergeben, daß eine Zu-1-Beziehung durch eine Zu-2- oder Zu-*x*-Beziehung ersetzt werden muß, und es ist nicht nachvollziehbar, warum diese Änderung in der Kardinalität zu weitreichenden Codeänderungen führen sollte. Die durch die Vereinheitlichung erlangte Uniformität beider Beziehungstypen führt somit dazu, dass das fundamentale Konzept der Beziehung hervorgehoben und den Vorteilen der jeweils für sich betrachteten Implementierung von Zu-1- bzw. Zu-*n*-Beziehungen nicht untergeordnet wird. Als besonders vorteilhafter Effekt ist neben der erleichterten Modifikation von Programmen zusätzlich das Entfallen des Problems der Null-Zeiger-Dereferenzierung zu nennen. Denn wie wir noch sehen werden, kann dieses lästige Problem durch entsprechende Einführung von Assoziationstypen auf elegante Weise umgangen werden. Dem vermeintlichen Nachteil des Verlustes des mittlerweile sehr eingeprägten Umgangs mit Zu-1-Beziehungen (durch einfache Dereferenzierung) kann durch eine entsprechende Programmiersprachenerweiterung, die diese Art des Zugriffes simuliert, entgegen gewirkt werden.

2.3 Verwandte Arbeiten

Ein bereits relativ früh entwickelter, in der Praxis jedoch wenig etablierter Ansatz macht sich die Entwicklung und den Einsatz von relationalen Datenbankprogrammiersprachen (Sprachen, in denen programmiersprachliche und datenbanksprachliche Konzepte miteinander verschmolzen werden), die das Relationenmodell in ihr Typsystem integrieren, zueigen. Bedeutende Arbeiten in diesem Bereich sind beispielsweise die relationalen Datenbankprogrammiersprachen Pascal/R [Sc77] und DBPL [SM92]. Pascal/R erweitert Pascal um relationale Datenbankkonstrukte, indem der Datentyp RELATION und entsprechende mengenorientierte Operationen (auf typisierten Relationenvariablen) zur Verfügung gestellt werden. Die Nachfolgesprache DBPL stellt eine vergleichbare Erweiterung der Programmiersprache Modula-2 dar. Beide Sprachen haben gemeinsam, das Konzept der Relation im Sinne einer Datenbankrelation (mitsamt der Definition von Schlüsseln) als eigenen Datentyp in eine höhere Programmiersprache zu integrieren. Die jeweilige Programmiersprache wird dabei in vergleichsweise hohem Maße erweitert.

Eine ganze Reihe von anderen Arbeiten (so z. B. [BGE07, BW05, NPN08, Øs07, Ru87]), von denen wir hier aus Platzgründen nur einige exemplarisch besprechen können, befaßt sich mit einer leichteren, nicht auf den Ersatz einer (zusätzlichen) Datenbanksprache, sondern auf die ausschließliche Integration von Relationen (im von Datenbanken abgekoppelten Sinne) fokussierten Erweiterung von objektorientierten Programmiersprachen. Als eine der ersten muß die von Rumbaugh aus dem Jahr 1987 genannt werden, in der, von der objektorientierten Modellierung herkommend, die Unverzichtbarkeit von Relationen für die semantische Nachbildung der Realität in Programmen besonders hervorgehoben wird [Ru87]. Zwar führt Rumbaugh Relationen als eine spezielle Art von Klassen ein, die instanziiert werden können (und deren Instanzen dann die Extension der Relation enthalten), jedoch erkennt er die Wichtigkeit der Navigation von Objekt zu Objekt an und schlägt deswegen vor, automatisch Methoden für die an einer Relation beteiligten Klassen zu generieren, die die Beziehung von einer Instanz der beteiligten Klassen zu denen der anderen Seite der Relation herstellen. Allerdings stützt sich die Implementierung dieser Methoden auf die Tupelmengen der Relationen, die separat (von den Objekten unabhängig) verwaltet werden.

An die Arbeit von Rumbaugh anknüpfend stellen Bierman und Wren ihre formal spezifizierte Sprache RelJ vor, die Relationen als Typen modelliert [BW05]. Anders als bei Rumbaugh sind Instanzen dieser Typen Tupel (und keine Tupelmengen), die von einer Relation unabhängig existieren können. RelJ sieht auch die Vererbung unter Relationen vor, jedoch mit einer eher fragwürdigen Semantik, auf die hier nicht weiter eingegangen werden kann. Daß die Relationen von RelJ gerichtet sind (und entsprechend nur in eine Richtung navigiert werden können), stellt zudem eine erhebliche Beschränkung dar.

Der Richtungscharakter der Relationen von RelJ bleibt in den Assoziationen von Østerbye nominal erhalten, wird jedoch durch unbeschränkte Zugriffsmöglichkeiten auf die (mit `To` und `From` gekennzeichneten) Assoziationsenden (und die daraus resultierende freie Navigierbarkeit) faktisch aufgehoben [Øs07]. Anders als [Ru87, BW05] schlägt Østerbye hierfür keine Spracherweiterung vor, sondern setzt auf eine Implementierung mittels Bibliotheken. Voll erhalten bleibt bei Østerbye die Unterscheidung von Zu-1- und Zu-*n*-Beziehungen, obwohl auch für erstere Assoziationen vorgesehen sind.

abstract data type *Association*

imports *Boolean*, *Card*, *Object*, *Collection*

syntax

new:	*Card*	$\rightarrow$ *Association*
bound:	*Association*	$\rightarrow$ *Card*
card:	*Association*	$\rightarrow$ *Card*
add:	*Association* $\times$ *Object*	$\rightarrow$ *Association* $\vee \perp$
add:	*Association* $\times$ *Collection*	$\rightarrow$ *Association* $\vee \perp$
remove:	*Association* $\times$ *Object*	$\rightarrow$ *Association*
remove:	*Association* $\times$ *Collection*	$\rightarrow$ *Association*
replace:	*Association* $\times$ *Object*	$\rightarrow$ *Association* $\vee \perp$
replace:	*Association* $\times$ *Collection*	$\rightarrow$ *Association* $\vee \perp$
contains:	*Association* $\times$ *Object*	$\rightarrow$ *Boolean*
collection:	*Association*	$\rightarrow$ *Collection*
object:	*Association*	$\rightarrow$ *Object* $\vee \perp$

semantics

$\forall\, n \in Card, a \in Association, o, o' \in Object, o \neq \text{null}, c \in Collection$:

$\max(\text{card}(a) - \text{card}(c), 0) \leq \text{card}(\text{remove}(a, c)) \leq \text{card}(a)$

$\text{card}(\text{add}(a, o)) = \perp$ **if** $\neg\text{contains}(a, o) \wedge \text{card}(a) = \text{bound}(a)$

$\text{card}(\text{add}(a, o)) \geq \text{card}(a) \vee \perp$ (von Subtyp festzulegen)

$\text{remove}(\text{add}(a, o'), o) = \text{add}(\text{remove}(a, o), o') \vee \perp$

$\text{contains}(a, o) = \text{card}(a) > \text{card}(\text{remove}(a, o))$

$\text{bound}(\text{new}(n)) = n$	$\text{card}(a) - 1 \leq \text{card}(\text{remove}(a, o)) \leq \text{card}(a)$
$\text{bound}(\text{add}(a, \_)) = \text{bound}(a) \vee \perp$	$\text{bound}(\text{replace}(a, \_)) = \text{bound}(a) \vee \perp$
$\text{add}(a, \text{null}) = a$	$\text{replace}(a, \text{null}) = \text{new}(\text{bound}(a))$
$\text{card}(\text{new}(n)) = 0$	$\text{replace}(a, o) = \text{add}(\text{new}(\text{bound}(a)), o)$
$\text{card}(\text{add}(\text{new}(n), o) = 1$	$\text{replace}(a, c) = \text{add}(\text{new}(\text{bound}(a)), c) \vee \perp$
$\text{card}(\text{add}(a, c)) \geq \max(\text{card}(a), \text{card}(c)) \vee \perp$	$\text{collection}(\text{add}(\text{new}(n), c)) = c \vee \perp$
$\text{bound}(\text{remove}(a, \_)) = \text{bound}(a)$	$\text{object}(\text{add}(\text{new}(n), o)) = o$ **if** $n = 1$ **else** $\perp$
$\text{remove}(a, \text{null}) = a$	$\text{remove}(\text{new}(n), o) = \text{new}(n)$

Abbildung 2: ADT *Association* zur einheitlichen Repräsentation von Zu-1- und Zu-*n*-Beziehungen. *Association* ist absichtlich unterspezifiziert (s. Text).

Andere Arbeiten haben sich nicht zum Ziel gemacht, die (objektorientierte) Programmierung zu erweitern, sondern befassen sich damit, wie (in der Regel aus Modellen stammende) Relationen in herkömmlichen objektorientierten Code abgebildet werden können [ABS04, Ge09]).

Weder das eine noch das andere verfolgt Microsoft mit seinem LINQ-Projekt: Hier wird vielmehr eine gemeinsame Schnittstelle für verschiedene Arten von Datenquellen definiert, die es erlaubt, SQL-ähnliche relationale Anfragen unabhängig von der Art einer Quelle zu formulieren [BMT07]. Der vermutlich interessanteste Beitrag ist hierbei die Vereinheitlichung des Zugriffs auf programminterne Collections und -externe Quellen wie relationale und XML-Datenbanken — die Abfragesprache selbst ist weniger revolutionär, insbesondere wenn man sich vor Augen hält, daß sie sich im wesentlichen auf Methoden, die λ-Ausdrücke als Parameter akzeptieren, zurückführen läßt (und damit nicht über das hinausgeht, was funktionale Sprachen und auch Smalltalk immer schon boten).

3 Beseitigung des Unterschieds von Zu-1- und Zu-*n*-Beziehungen

Für die Umsetzung von Zu-*n*-Beziehungen haben Collections in der objektorientierten Programmierung eine zentrale Bedeutung. Gegenüber den immer gleich gearteten Relationen des Relationenmodells besitzen sie den Vorteil, daß man ihnen, da sie als ganz normale Klassen implementiert sind, beliebiges Verhalten beiordnen kann. So lassen sich leicht geordnete oder gar sortierte Beziehungen definieren (also Beziehungen, in denen die Elemente auf der *n*-Seite eine feste Reihenfolge haben oder sortiert sind; in Smalltalk beispielsweise Ordered oder Sorted Collections), es lassen sich Elemente durch einen Schlüssel gezielt auffinden (Arrays oder Dictionaries), es läßt sich festlegen, ob ein Objekt einfach oder mehrfach in einer Beziehung zum Ausgangsobjekt stehen kann (Sets oder Bags) und so weiter. Dieser Vorteil sollte nicht aufgegeben werden.

Es bleibt also nur, wenn man die grundlegende Unterscheidung von Zu-1- und Zu-*n*-Beziehungen beseitigen will, das programmiersprachliche Konstrukt für Zu-1-Beziehungen anzugleichen. Da Zu-1-Beziehungen ein Spezialfall von Zu-*n*-Beziehungen sind, ist dies kein theoretisches Problem — ein praktisches hingegen schon, zumindest wenn man für eine gewisse Akzeptanz unter Programmierern sorgen will, denen es ja heute schon freisteht, Zu-1-Beziehungen mittels (einelementiger) Collections umzusetzen, die dies aber schon aufgrund des zusätzlichen Programmieraufwands wohl kaum freiwillig tun würden. Es geht hier also zunächst um die geschickte Wahl einer einheitlichen Syntax.

3.1 Definition eines abstrakten Datentypen für die einheitliche Behandlung von Zu-1- und Zu-*n*-Beziehungen

Abbildung 2 stellt eine solche Syntax in Form der Definition eines abstrakten Datentypen (ADT) *Association* vor. Die Definition von *Association* stützt sich auf ein paar andere, als gegeben vorausgesetzte Datentypen: Der Datentyp *Boolean* ist Standard, *Card* entspricht den natürlichen Zahlen ergänzt um ein Element „∞“ für „beliebig“, das größer ist als jede Zahl aus *Card*, *Object* ist der Datentyp beliebiger Objekte (die natürlich selbst typisiert sind; die Typisierung lassen wir hier aber unberücksichtigt) und *Collection* der von beliebigen Collections (wir setzen hier lediglich die Existenz der Operationen add und card mit üblicher Syntax und Semantik voraus). „null“ ist ein Wert vom Typ *Object* mit der üblichen Bedeutung. „⊥“ benennt einen Funktionsausdruck, dessen Ergebnis undefiniert ist und der somit zu einem Fehler führt. Wie üblich vererben sich Fehler in dem Sinne, daß ein Funktionsausdruck, der einen undefinierten Funktionsausdruck enthält, ebenfalls undefiniert ist.

Die Funktionen add, remove und replace können in konkreter (Programmiersprachen-) Syntax durch die Infix-Operatoren `+=`, `-=` bzw. `:=` ersetzt werden. Sie sind in der zweiten Operandenstelle überladen; der Unterstrich an der Stelle steht für einen beliebigen Operanden eines der für die Stelle zulässigen Typen (*Object* oder *Collection*). replace ersetzt die Zuweisung (doch Achtung: Durch `a := null` wird nur der Inhalt der Assoziation `a` ersetzt und nicht `a` selbst!) und ist im Falle von Zu-1-Beziehungen vermutlich die am häufigsten verwendete Operation. Die Operatoren collection und object dienen der Verwendung von *Association*-Objekten in Ausdrücken, die eine Collection bzw. ein Objekt er-

warten, so z. B. in For-each-Schleifen oder bei einem Test auf Gleichheit mit einem Objekt. Man beachte jedoch, daß hier die Unterscheidung zwischen Zu-1- und Zu-*n*-Beziehungen wieder eingeführt wird: Die Anwendung von object auf einem *Association*-Objekt, dessen Kardinalität als > 1 angegeben wurde, führt zu einem Fehler.

Dem aufmerksamen Leser wird aufgefallen sein, daß *Association* unterspezifiziert ist. So ist beispielsweise nicht klar, was passiert, wenn einer Instanz von *Association* ein Objekt hinzugefügt wird, das sie schon enthält. Dies ist Absicht und liegt darin begründet, daß mit *Association* nicht festgelegt sein soll, ob es sich beim Inhalt um Mengen, Multimengen, Listen oder was auch immer handeln soll. Da aber keiner dieser konkreten Typen ein Supertyp aller anderen ist, ist hier *Association* gewissermaßen als abstrakter ADT definiert, was soviel heißen soll wie daß es keine Implementierungen gibt, die genau seiner Spezifikation entsprechen, d. h., die keine zusätzlichen funktionalen Eigenschaften haben.

3.2 Implementierung und Verwendung des ADT *Association* in objektorientierten Programmiersprachen

Der ADT *Association* ist analog zur Wurzel Collection eines Collection-Frameworks wie dem Smalltalks oder Javas (in *Association* durch den ADT *Collection* repräsentiert) zu sehen, was soviel heißt wie daß konkrete Implementierungen Eigenschaften hinzufügen und insbesondere einen Typparameter (für den Typ der bezogenen Objekte, hier durch *Object* vertreten) haben können. *Association* wird also typischerweise als abstrakte Klasse implementiert, von der andere, konkrete ableiten. Da wir hier aber an Implementierungsdetails (und auch an den zahlreichen Erweiterungsmöglichkeiten, die denkbar sind) nicht interessiert sind, nehmen wir im folgenden an, daß Association eine konkrete Klasse ist, die über alle Operationen des ADT *Association* verfügt. Genau wie Collection in Java, C# und anderen Sprachen sei Association dem Compiler bekannt, so daß er bestimmte syntaktische und semantische Tests durchführen sowie speziellen Code generieren kann.

Da sie der Umsetzung von Beziehungen dienen, können nur Instanzvariablen (Felder) mit *Association*-Typen (konkreten Implementierungen unseres ADT *Association*) deklariert werden. Wir nennen diese Instanzvariablen dann *Assoziationen* und unterscheiden sie fortan von den anderen Instanzvariablen sorgfältig. Anders als normale Instanzvariablen sind Assoziationen nämlich nicht zuweisbar — sie können also weder auf der linken Seite einer Zuweisung noch als formale Parameter von Methoden auftreten..

Bei der Deklaration einer Assoziation soll nicht der Containertyp (Association<Angestellter> angestellte = new Association<Angestellter>();) sondern der Elementtyp (Angestellter(<x>) angestellte = new Association<Angestellter>(); wobei <x> für die Kardinalität steht) im Vordergrund stehen. Dies ist auch insofern gerechtfertigt, als, wie wir gleich sehen werden, der genaue Containertyp, Association oder ein Subtyp davon, außer bei der Initialisierung von Assoziationen keine Rolle spielt — Assoziationen können daher immer vom Typ Association angenommen werden.

Über die Elemente einer Assoziation kann dann (per impliziter Konversion in eine Collection) mittels foreach iteriert werden — es werden der Laufvariable der Reihe nach die

Elemente, die mit dem Besitzer der Assoziation über diese in Beziehung stehen, zugewiesen. Einen weiteren Vorteil stellt dar, dass mit Hilfe von integrierten Abfragesprachen wie Microsofts LINQ Abfragen auf Zu-1-Beziehungen ausgedehnt werden können. Damit zusammenhängend tritt zudem der Vorteil zutage, daß bei Ergebnissen derartiger Abfragen „kein Ergebnis“ durch eine leere Menge repräsentiert wird und somit in der Regel keiner Sonderbehandlung (wie einer Prüfung auf not null) bedarf; das Problem der Null-Zeiger-Dereferenzierung in Bezug auf Zu-1-Beziehungen entfällt somit.

4 Einführung von Bidirektionalität

4.1 Spezifikation mit abstrakten Datentypen

Eine bidirektionale Beziehung als aus zwei unidirektionalen zusammengesetzt zu implementieren ist nichts grundsätzlich Schlechtes — es kommt nur darauf an, den Programmierer von der Verantwortung zu befreien, die beiden zu koordinieren. Um dies zu erreichen, führen wir zunächst einen ADT *Relation* ein, der Relationsdeklarationen spezifiziert. Den Konstruktor von Relation nennen wir declare, um auszudrücken, daß er nicht irgendwann im Programmablauf, sondern, eben als Deklaration, bereits bei der Übersetzung ausgewertet wird. Neben den Argumenttypen (den Typen, die den Stellen der Relation zugeordnet sind und die deren Herkunft bestimmen), die wir hier weglassen und implizit als *Object* annehmen[2], verlangt der Konstruktor (die Deklaration) die Angabe zweier Funktionen, die jeweils ein Objekt auf eine Assoziation abbilden (und zwar genau die, die die Relation für das Objekt in Richtung auf seine Gegenüber realisiert). Diese Funktionen werden selbst durch einen ADT repräsentiert, den wir (zugegebenermaßen etwas unbeholfen) *Role* genannt haben. Die Syntax beider Datentypen sowie die Semantik von *Relation* sind in Abbildung 3, oben, zu sehen; die Semantik von *Role* wird im Kontext seiner Verwendung weiter unten definiert werden. Es sei jedoch hier schon bemerkt, daß wir dabei anstelle von *apply*(*r*, *o*) wie für Funktionsanwendungen üblich *r*(*o*) schreiben werden. Daß wir hier überhaupt auf eine Funktion höherer Ordnung zurückgreifen müssen, deckt sich damit, daß die Implementierung nicht ohne Reflektion (Metaprogrammierung) auskommt (s. Abschnitt 5).

Als letztes benötigen wir noch einen Datentyp, der die Pflege von bidirektionalen Beziehungen erlaubt. Im Falle unidirektionaler Beziehungen geschah dies ja mittels des ADT *Association*, der jedoch wie oben erläutert hier nicht mehr ausreicht, da ihm die für die Pflege der Rückrichtung benötigten Parameter fehlen. Um den Anwendungscode soweit wie möglich von der Gerichtetheit der Relationen unabhängig zu machen, definieren wir einen neuen ADT *HalfRelation* mit bis auf den Konstruktor zu *Association* identischer Syntax (Abbildung 3, unten). Auch die Semantik von *HalfRelation* basiert wesentlich auf dem ADT *Association*, für den er damit eine Art Wrapper bildet.

[2] Wir hätten sie (wie zuvor auch schon bei *Association*) als Typparameter der Definition des ADT angeben können; da sie aber für unsere weiteren Betrachtungen keine Rolle spielen, lassen wir sie weg.

abstract data type *Relation*
imports *Role*
syntax
declare: $Role \times Role \rightarrow Relation \vee \bot$
counterrole: $Relation \times Role \rightarrow Role \vee \bot$
semantics
$\forall\ r, s, t \in Role, r \neq s \neq t \neq r$:
declare $(r, r) = \bot$
counterrole(declare $(r, s), t) = \bot$
counterrole(declare $(r, s), r) = s$
counterrole(declare $(r, s), s) = r$

abstract data type *Role*
imports *Object*, *Association*
syntax
new: $\rightarrow Role$
apply: $Role \times Object \rightarrow Association \vee \bot$

abstract data type *HalfRelation*
imports *Relation*, *Role*, *Association*, *Object*
syntax
new: $Object \times Relation \times Role \times Association \rightarrow HalfRelation$
add: $HalfRelation \times Object \rightarrow HalfRelation$
remove: …
semantics
$\forall\ a, b \in Object, r \in Role, R \in Relation$:
$\langle$add(new(a, R, r, $r(a)$), b); new(b, R, counterrole(R, r), counterrole(R, r)(b))$\rangle$ =
$\langle$new(a, R, r, add($r(a)$, b)); new(b, R, counterrole(R, r), add(counterrole(R, r)(b), a))$\rangle$
remove …

Abbildung 3: ADTs *Relation*, *Role* und *HalfRelation* zur Repräsentation bidirektionaler Beziehungen. Die Definition von *HalfRelation* folgt ab *remove* analog zu *Association* (Abbildung 2).

Im Gegensatz zu dem von *Association* verlangt der Konstruktor von *HalfRelation*, new, die Angabe des besitzenden Objekts, also des Objekts, von dem die Assoziation ausgeht. Man beachte, daß dieses Objekt, a, im Fall des Ausdrucks add($a.f$, b) zwar im Kontext bekannt ist, nicht aber der Assoziation f, der b hinzugefügt werden soll. Dies wird durch die Einführung von *HalfRelation*, dessen Instanz f' neben einer Assoziation f auch ihren Besitzer a kennt, korrigiert: add($a.f'$, b) delegiert das Hinzufügen zur eigenen Richtung an die zur Halbrelation f' gehörende Assoziation f und das Hinzufügen der Rückrichtung an die entsprechende Assoziation von b. Damit letztere bestimmt werden kann, verlangt new weiterhin die Angabe einer Relationsdeklaration R und einer Rolle r; die Funktionsanwendung counterrole(R, r) liefert dann eine Funktion, die, auf b angewendet, die Assoziation der Rückrichtung liefert, der dann a hinzugefügt wird. Mit anderen Worten: Für $a.f'$ = new(a, R, r, f) mit $f \in Association$ führt add($a.f'$, b) zu add(f, b) und zu add(counterrole(R, r)(b), a). Man beachte, daß die axiomatische Definition der Semantik von add die Identität von $r(a)$ und f im obigen Beispiel ausnutzt.

4.2 Übertragung auf die objektorientierte Programmierung

Um eine bidirektionale Beziehung in einem objektorientierten Programm herzustellen, deklarieren wir nun einfach Instanzvariablen mit Klassen, die den ADT *HalfRelation* implementieren. Gegenüber dem Einsatz von unidirektionalen Beziehungen ändert sich der

Code nur an den Stellen der Deklarationen, bei denen nun HalfRelation anstelle von Association stehen muß, wobei der Konstruktoraufruf einen Parameter erhält, der die Relation benennt (wir setzen wieder die Existenz einer konkreten Klasse HalfRelation voraus):

```
Arbeitsplatz(1) arbeitsplatz = new HalfRelation<Arbeitsplatz>(Sitzt);
Angestellter(2) angestellter = new HalfRelation<Angestellter>(Sitzt);.
relation Sitzt(Angestellter.arbeitsplatz, Arbeitsplatz.angestellter)
```

wobei arbeitsplatz in der Klasse Angestellter, angestellter in der Klasse Arbeitsplatz und die Relation Sitzt im Programm deklariert wird.

Man beachte, daß bis auf die Angabe der Relation sämtliche Information, die zur Instanziierung einer Halbrelation benötigt wird, vom Compiler aus dem Kontext gewonnen werden kann: Das besitzende Objekt steckt in this, die Rolle ergibt sich aus dem Namen der Instanzvariable, die die Halbrelation benennt (der sie im Rahmen der Deklaration zugewiesen wird), und die Assoziation wird im Konstruktor neu erzeugt. Die syntaktische Last für den Programmierer ist also ausgesprochen gering.

5 Prototypische Implementierung in C#

In einem ersten Ansatz haben wir die oben beschriebenen ADTs *Association* und *HalfRelation* in parametrisierte C#-Bibliotheksklassen umgesetzt, die der Deklaration entsprechender Instanzvariablen und der Erzeugung ihrer Inhalte dienen. Nicht umsetzen konnten wir damit die Bedingungen, daß *nur* Instanzvariablen mit ihnen deklariert, daß diese Instanzvariablen keinen anderen Variablen zugewiesen (auch nicht vom Typ Object) und daß ihnen selbst nach der Initialisierung keine weiteren Werte zugewiesen werden dürfen (letzteres kann aber zumindest durch Verwendung des C#-Modifiers readonly bei der Instanzvariablendeklaration erzwungen werden). Auch die Ableitung der für die Instanziierung einer Halbrelation notwendigen Information aus dem Kontext konnten wir auf diese Weise nicht umsetzen — sie muß beim Konstruktoraufruf explizit übergeben werden. Anstelle einer Relationsdeklaration wie oben beschrieben übergeben wir beim Konstruktoraufruf von HalfRelation neben this den Namen der Relation als String. Bei Aktualisierungen von Instanzen von HalfRelation wird dann anhand des hinzugefügten oder weggenommenen Objekts die Klasse der Gegenseite ermittelt und dort, via Reflektion, nach der Instanzvariable gesucht, die eine Halbrelation mit gleichem Relationsnamen enthält (und die — bei homogenen Relationen — von der Ausgangshalbrelation verschieden ist). Die Aktualisierung wird dann auf der Gegenseite entsprechend durchgeführt.

Insgesamt kann diese prototypische Implementierung nur als erster Versuch einer Umsetzung betrachtet werden. Als ein auf dieser Arbeit aufbauendes und in naher Zukunft umzusetzendes Ziel ist die diesbezügliche Erweiterung objektorientierter Programmiersprachen und die Durchführung von empirischen Untersuchungen zur Performance, Wartbarkeit etc. zu betrachten. Hierbei sollte das Augenmerk insbesondere auf die Darstellung bidirektionaler Pointermanipulationen als atomare Anweisungen gelegt werden.

6 Zusammenfassung und Schluß

Wir machen die objektorientierte Programmierung relationaler, indem wir die herkömmliche Umsetzung von Zu-1- und Zu-*n*-Beziehungen über direkte Zeiger bzw. Collections vereinheitlichen. Zu diesem Zweck führen wir Assoziationen ein, die den indizierten Instanzvariablen Smalltalks insoweit gleichen, als sie Beziehungen zu mehreren anderen Objekten gleichzeitig herzustellen erlauben, ohne dabei (wie Collections) von ihrem Objekt getrennt werden zu können, die aber im Gegensatz zu den indizierten Instanzvariablen Smalltalks benannt sind, so daß ein Objekt mehrere haben kann. Indem wir zusätzlich Relationen als Paare von korrespondierenden Assoziationen einführen, ermöglichen wir auch noch die vollautomatische Pflege von bidirektionalen Beziehungen. Im Zusammenspiel mit relationalen Abfragesprachen wie LINQ, die auf Collections operieren können, ergibt sich somit eine relationale Sicht auf objektorientierte Daten, die trotzdem den navigierenden (d. h. zeigerbasierten) Charakter der objektorientierte Programmierung vollständig erhält, die also insbesondere ohne Relationen als von Objekten separat zu verwaltende Tupelmengen auskommt. Damit glauben wir, die Vorzüge beider Weltanschauungen in einer gemeinsamen vereint zu haben.

Literaturverzeichnis

[ABS04] Amelunxen, C.; Bichler, L.; Schürr, A.: Codegenerierung für Assoziationen in MOF 2.0.Modellierung 2004. LNCS, vol. P-45. Springer-Verlag, 2004, S. 149-168.

[BGE07] Balzer, S.; Gross, T.R.; Eugster, P.: A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. ECOOP 2007.LNCS, Springer, S. 323-346.

[BMT07] Bierman, GM; Meijer, E.; Torgersen, M.: Lost in translation: formalizing proposed extensions to C#. SIGPLAN Not. 42, 10 (Oct. 2007), 479–498.

[BW05] Bierman, G.; Wren, A.: First-Class Relationships in an Object-Oriented Language. ECOOP 2005. LNCS, vol. 3586. Springer-Verlag, 2005, S. 25-29.

[CM84] Copeland, C.; Maier, D.: Making smalltalk a database system. In ACM SIGMOD Records, vol. 14, 2, 1984, S. 316-325.

[Ge09] Gessenharter, D: Implementing UML associations in Java: a slim code pattern for a complex modeling concept. RAOOL '09. ACM Press, 2009, S. 17-24.

[NPN08] Nelson, S.; Noble, J.; Pearce, D.J.: Implementing first-class relationships in Java. In Proceedings of RAOOL'08. ACM Press, 2008.

[Øs07] Østerbye, K.: Design of a class library for association relationships. In Proceedings of LCSD '07. ACM Press, 2007, S. 67-75.

[Ru87] Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. OOPSLA '87. ACM Press, 1987, S. 466-481.

[Sc77] Schmidt, J.W.: Some high level language constructs for data of type relation. In ACM Transactions on Database Systems, Vol.2, No.3, 1977, S. 247-261.

[SM92] Schmidt, J.W.; Matthes, F.: The database programming language DBPL – Rationale and Report. Technical Report FIDE/92/46, FB Informatik, Universität Hamburg, 1992.

[SS09] Stadtler, D.; Steimann, F.: Wie die Objektorientierung relationaler werden sollte — Eine Analyse aus Sicht der Datenmodellierung. eingereicht bei: Modellierung 2010 (von den Autoren erhältlich).

Formale Semantik modularer Zeitverfeinerung in AUTOFOCUS.

David Trachtenherz
Institut für Informatik, Technische Universität München,
Boltzmannstr. 3, D-85748, Garching bei München, Germany
trachten@in.tum.de

Abstract: Moderne automobile eingebettete Systeme bilden hochkomplexe verteilte Steuergerätenetzwerke. Modellbasierte Entwicklung ist ein verbreiteter Ansatz, um dieser Komplexität zu begegnen. AUTOFOCUS ist ein CASE-Werkzeugprototyp zur formal fundierten modellbasierten Entwicklung eingebetteter Systeme. Ein Modell wird hierarchisch aus Komponenten aufgebaut, die über getypte Kanäle kommunizieren. AUTOFOCUS verfügt über eine robuste und übersichtliche formale Semantik: die Kommunikation und Ausführung sind über einen globalen Takt für alle Komponenten synchronisiert. In diesem Artikel stellen wir eine Erweiterung der AUTOFOCUS-Semantik um eine Zeitverfeinerung von Komponenten vor. Diese ermöglicht eine einfachere Modularisierung von Systemen, eine bessere Strukturierung von Verhaltensspezifikationen und eine bessere Abbildung auf die Zielplattform realer eingebetteter Systeme. Gleichzeitig bleiben die Vorteile fest getakteter AUTOFOCUS-Semantik wie starke Kausalität und sichere Terminierung von Berechnungsschritten erhalten.

1 Einleitung und Motivation

Die Bedeutung softwarebasierter Funktionalitäten für automobile eingebettete Systeme nimmt beständig zu. Spielte die Software zu Beginn der 1990er Jahre noch keine wesentliche Rolle, so sind eingebettete Softwaresysteme aus modernen Automobilen nicht mehr wegzudenken: Die Elektronik/Software macht bereits ein Drittel der Herstellkosten eines Fahrzeugs aus, wobei ein wesentlicher Anteil auf Software entfällt; Die Innovationen werden zu 80-90% von der Software getrieben; In modernen Oberklassefahrzeugen kann der Softwareumfang 1 GB erreichen. Entsprechend ist die Bedeutung der Software für den Automobilbau in wissenschaftlichen wie industriellen Veröffentlichungen anerkannt [PBKS07, Her06, Fri03, Gri03, Rei06].

Die Bereitstellung der Entwicklungstechniken für umfangreiche und gleichzeitig hochqualitative eingebettete Systeme ist eine der wichtigen Herausforderungen der Softwaretechnik. Modellbasierte Entwicklungstechniken werden seit geraumer Zeit bei der Entwicklung der Steuergerätesoftware im Automobilbau eingesetzt, aber auch in anderen Domänen mit hohen Zuverlässigkeitsanforderungen, wie beispielsweise Avionik. Mittlerweile existieren zahlreiche modellbasierte Entwicklungswerkzeuge, darunter MATLAB/Simulink/Stateflow, ASCET, SCADE [ABR05, ASC, SCA]. Ein wichtiges Kennzeichen modellbasierter Techniken ist, dass ein System als Netzwerk miteinander kommunizierender

und hierarchisch aufgebauter Komponenten dargestellt wird. Solche Techniken bieten insbesondere den Vorteil anschaulicher graphischer Beschreibung der Systemstruktur durch Komponentendiagramme. Außerdem unterstützen sie häufig die strukturierte Verhaltensbeschreibung durch Zustandsautomaten.

Zu den wesentlichen Merkmalen einer Modellierungstechnik gehören die Fragen, wie Komponenten miteinander kommunizieren, welche Techniken zur Verhaltensbeschreibung verfügbar sind, wie größere Systeme dekomponiert werden können und, nicht zuletzt, ob es eine präzise definierte Semantik gibt. Der letzte Punkt ist insofern kritisch, als es häufig keine eindeutige formale Semantik (oder zumindest keine veröffentlichte) gibt, oder mehrere unterschiedliche Semantikvarianten existieren, wie z. B. für Statecharts [vdB94]. Dies kann dazu führen, dass das gleiche Modell auf unterschiedliche Weise von Entwicklern, aber auch weiterverarbeitenden Werkzeugen, wie Codegeneratoren, interpretiert wird. Entsprechend erlaubt eine präzise und gleichzeitig möglichst einfach gehaltene Semantikdefinition eine eindeutige Interpretation von Modellen und vereinfacht die Entwicklung sowie Weiterverarbeitung der Modelle im Laufe des Systementwicklungsprozesses.

Ein weiterer wichtiger Aspekt einer Modellierungstechnik ist die Realisierung modularer Dekomposition des Gesamtsystems zu Teilsystemen/Komponenten – dies ist von entscheidender Bedeutung, da die Komplexität und Qualität realer Systeme durch Aufteilung in kleinere Teilsysteme mit definierten Schnittstellen besser beherrscht werden können.

Diese Überlegung findet ihren Niederschlag auch in dem AUTOFOCUS-Werkzeug, das als Prototyp zur modellbasierten Entwicklung in der Domäne eingebetteter Systeme konzipiert wurde [HSSS96, HSE97]. Es erlaubt die graphische Darstellung der Systemstruktur sowie der Zustandsautomaten, und verfügt über eine einfache und robuste formale Semantik [Tra09b, Abschnitt 4.1.2] [Tra09a, Abschnitt 4.2].

Alle Komponenten in einem AUTOFOCUS-Modell laufen bisher mit gleicher Geschwindigkeit, was den Nachteil mit sich bringt, dass zum einen die modulare Dekomposition in Teilsysteme und die Strukturierung von Zustandsautomaten eingeschränkt werden, und zum anderen die Zielplattform verteilter eingebetteter Systeme mit unterschiedlich schnellen Teilsystemen (z. B. Steuergeräten) nicht hinreichend realistisch abgebildet wird – dies wird in Abschnitt 4 ausführlich erläutert. In diesem Artikel stellen wir eine einfache und pragmatische Erweiterung der AUTOFOCUS-Semantik um die modulare Zeitverfeinerung vor, die diese Einschränkungen weitgehend behebt, indem für jede Komponente ein fester Taktfaktor angegeben werden kann, um den sich die interne Ausführungsgeschwindigkeit von der Umgebung unterscheidet. Im Weiteren führen wir aus, wie die neue Semantik die Modellierungsmöglichkeiten in AUTOFOCUS erweitert, so dass nun Modelle ähnlich flexibel wie in verschiedenen gängigen Modellierungstechniken entwickelt werden können, und dabei die Vorteile einer übersichtlichen Semantik erhalten bleiben.

Der Artikel gliedert sich wie folgt: der Abschnitt 2 stellt Grundbegriffe der AUTOFOCUS-Modellierungstechnik kurz vor; in 3 wird die herkömmliche AUTOFOCUS-Semantik beschrieben; anschließend erörtert 4 die Erweiterung der Semantik um modulare Zeitverfeinerung durch die Mehrtaktsemantik; im nachfolgenden Abschnitt 5 werden die Auswirkungen modularer Zeitverfeinerung erörtert; in Abschnitt 6 werden verwandte Ansätze besprochen und schließlich zieht Abschnitt 7 das Fazit.

2 Grundbegriffe

Zu Beginn stellen wir die Grundlagen der Modellierung in AUTOFOCUS vor. Die Modellierungstechnik basiert auf Ausschnitten der FOCUS-Spezifikationstechnik [BS01]. Ein Modell wird hierarchisch als Strukturbaum aus *Komponenten* aufgebaut:

- **Verhaltensspezifikation durch Zustandsautomaten**: Die Blätter eines Modellstrukturbaums sind atomare Komponenten, deren Verhalten durch ein Zustandsübergangsdiagramm, d. h. Eingabe-/Ausgabeautomaten definiert wird.
- **Strukturspezifikation durch hierarchische Dekomposition**: Die Knoten eines Modellstrukturbaums sind zusammengesetzte Komponenten, die in Teilkomponenten aufgeteilt sind.

Komponenten können beliebig viele getypte *Eingabe- und Ausgabeports* besitzen. Die Kommunikation zwischen Teilkomponenten einer zusammengesetzten Komponente erfolgt über gerichtete *Kanäle*, die Nachrichten von Ausgabe- zu Eingabeports übertragen.

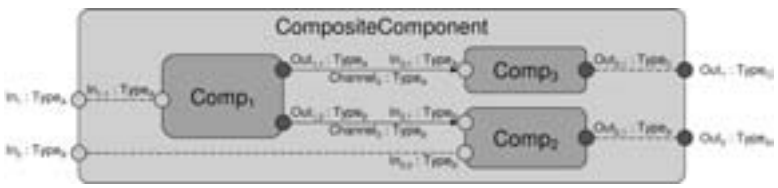


Abbildung 1: Strukturdiagramm – Zusammengesetzte Komponente

Die Komposition ist einfach und intuitiv: Eine (Ober-)Komponente kann sich aus mehreren Teilkomponenten zusammensetzen (Abb. 1). Kommunikationsports der Teilkomponenten können den Ports der Oberkomponente zugeordnet werden – dadurch gelangen Nachrichten von der Komponentenschnittstelle zu den Teilkomponenten und umgekehrt (z. B. die Portpaare $(In_1, In_{1,1})$ und $(Out_{2,1}, Out_2)$ in Abb. 1). Die Ports der Teilkomponenten können miteinander über Kanäle verbunden werden (z. B. das Portpaar $(Out_{1,1}, In_{3,1})$).

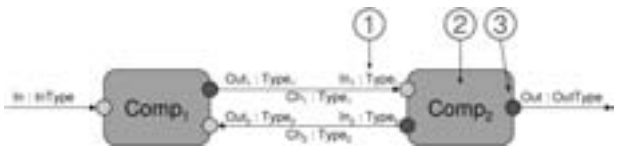


Abbildung 2: Teilschritte eines Berechnungsschritts

Die Berechnung einer Komponente ist eine i. A. unendliche Abfolge von Rechenschritten. Ein Rechenschritt zu einem Zeitpunkt $t \in \mathbb{N}$ besteht aus folgenden Teilschritten (Abb. 2):

1) **Einlesen der Eingaben**: Die im vorherigen Schritt $t-1$ produzierten Ergebnisse werden übertragen. Falls an einem Ausgabeport keine Nachricht ausgegeben wurde (auch für $t = 0$ der Fall, d. h. zu Beginn der Berechnung), wird die leere Nachricht ε als Eingabe übermittelt.

2) **Berechnung**: Alle Komponenten führen ihre Berechnungsschritte durch. Die eingelesene Eingabe und der interne Komponentenzustand werden zur Berechnung der Ausgabe verwendet.

3) **Ausgabe**: Die berechneten Ausgabewerte werden an die Ausgabeports geschrieben. Für die Umgebung sind die Ergebnisse erst im nächsten Schritt $t+1$ sichtbar, als sie im Teilschritt (1) übertragen werden. Diese Kommunikationssemantik stellt starke Kausalität sicher, d. h., Ausgaben können nicht in dem gleichen, sondern erst im nachfolgenden Rechenschritt verwendet werden.

Dieser Ablauf beschreibt das Verhalten einer Komponente in einem Kommunikationstakt. Das Verhältnis zwischen der externen Kommunikation und den internen Berechnungen, das den wesentlichen Unterschied zwischen Eintakt- und Mehrtaktsemantik bildet, wird in den nächsten Abschnitten beschrieben.

3 Eintaktsemantik – Globaler Gleichtakt

In der herkömmlichen Eintaktsemantik werden alle Komponenten mit dem gleichen Takt sowohl für die Kommunikation als auch für interne Berechnungen betrieben. Wie in Abschnitt 2 beschrieben, kann das Verhalten einer Komponente C auf zwei Weisen definiert werden: für eine atomare Komponente wird das Verhalten direkt durch ein Zustandsübergangsdiagramm spezifiziert; für eine zusammengesetzte Komponente ergibt sich das Verhalten aus dem Verhalten ihrer Teilkomponenten und der internen Kommunikationsstruktur (Kommunikationsverbindungen unter den Teilkomponenten sowie Zuordnungen der Kommunikationsports von Teilkomponenten zu Kommunikationsports der (Ober-)Komponente C). Deshalb genügt die Definition der Semantik atomarer Komponenten und zusammengesetzter Komponenten zur Definition der AUTOFOCUS-Semantik.

Atomare Komponenten Zur Verhaltensspezifikation einer atomaren Komponente wird ein Eingabe-/Ausgabeautomat verwendet. Ein Automat enthält eine endliche Anzahl von Kontrollzuständen sowie Transitionen zwischen ihnen und kann zusätzlich lokale Variablen verwenden. Jede Transition hat neben dem Zielzustand vier Bestandteile:

- Eingabemuster: Das *Eingabemuster* definiert, an welchen Eingabeports nicht-leere Nachrichten anliegen müssen und weist ihren Werten lokale Bezeichner zu. Für einen Port p bezeichnet das Eingabemuster $p?v$, dass er nicht-leer sein muss und auf den Eingabewert mit v zugegriffen werden kann, während das Eingabemuster $p?$ bestimmt, dass p leer sein muss. Taucht ein Eingabeport in dem Eingabemuster der Transition nicht auf, so wird sein Eingabewert nicht berücksichtigt.
- Vorbedingung: Die *Vorbedingung* definiert eine boolesche Bedingung auf den Werten der Eingaben und ggf. der lokalen Variablen, die zutreffen muss, damit die Transition aktiviert wird.
- Ausgabemuster: Das *Ausgabemuster* legt fest, welche Werte an die Ausgabeports geschrieben werden, falls die Transition schaltet. Für einen Port p bezeichnet das Ausgabemuster $p!expr$, dass der Ausdruck $expr$ ausgewertet und das Ergebnis auf p ausge-

geben wird. Ein Ausgabemuster $p!$ bedeutet, dass keine Ausgabe erfolgt ($p = \varepsilon$). Den gleichen Effekt hat das Weglassen eines Ports im Ausgabemuster der Transition.

- Nachbedingung: In der Nachbedingung können lokalen Variablen neue Werte zugewiesen werden. Eine Zuweisung $v = expr$ bedeutet, dass der Ausdruck $expr$ ausgewertet und das Ergebnis der lokalen Variablen v zugewiesen wird. Kommt eine lokale Variable in der Nachbedingung in keiner Zuweisung auf der linken Seite vor, so behält sie ihren alten Wert (zu Beginn der Berechnung wird jeder lokalen Variablen ein Initialwert zugewiesen, der beliebig gewählt werden darf).

Betrachten wir als Beispiel die Transition

$$p_1?x; p_2?y \ : \ x < y \ : \ cmpRes! - 1 \ : \ lastDiff = y - x$$

zwischen Zuständen q_i und q_j. Sie kann schalten, wenn die Eingabeports p_1, p_2 nicht-leer sind und die Eingabe an p_1 kleiner als an p_2 ist. In diesem Fall wird an dem Ausgabeport $compRes$ der Wert -1 ausgegeben und in der lokalen Variablen $lastDiff$ die Differenz zwischen den Eingaben an p_2 und p_1 gespeichert.

Befindet sich der Automat in einem Kontrollzustand q_i, so besteht ein Berechnungsschritt in der Auswahl und Ausführung einer Transition aus dem Zustand q_i, deren Eingabemuster und Vorbedingung erfüllt sind. Ist keine solche Transition vorhanden, so bleibt der Automat im Zustand q_i und liefert leere Ausgaben an allen Ausgabeports. Sind mehrere Transitionen aktiviert, so kann jede von ihnen schalten, was zu Nichtdeterminismus führt (dieser wird bei Bedarf, z. B., für Code-Generierung, durch eine totale Ordnung auf Transitionen aufgelöst [Tra09a]).

Eine vollständige formale Semantikdefinition für Zustandsübergangsdiagramme findet sich in [Tra09a, Abschnitt 4.2].

Zusammengesetzte Komponenten Die Semantik einer zusammengesetzten Komponente ergibt sich aus den Semantiken der Teilkomponenten und der Kommunikationsstruktur. Seien $C_1, \ldots, C_n$ Teilkomponenten einer Komponente C. Die Kommunikation zwischen den Teilkomponenten findet, wie in Abschnitt 2 beschrieben, im ersten der drei Teilschritte eines Berechnungsschritts beim Einlesen der Eingaben statt. Die Kommunikationssemantik bestimmt für jeden Berechnungsschritt die Werte für jeden Eingabe- und Ausgabeport in einer zusammengesetzten Komponente. Für einen Eingabeport p sind folgende Fälle zu unterscheiden:

- Ist p Eingabeport der Wurzelkomponente des Modells, d. h., gehört er zur Umgebungsschnittstelle des Modells/Systems, so wird sein Wert der Eingabe aus der Umgebung (Eingabe an der Systemschnittstelle) zum Zeitpunkt t entnommen.
- Ist p Eingabeport einer Teilkomponente C_i und einem Eingabeport p_{in} der Oberkomponente C zugewiesen, so ist sein Wert gleich dem Wert an p_{in}: $\forall t : \ p(t) = p_{in}(t)$.
- Ist p Eingabeport einer Teilkomponente C_i und über einen Kanal mit einem Ausgabeport p_{out} einer Teilkomponente C_j verbunden (wobei auch $i = j$ zugelassen ist) so ist sein Wert gleich dem Wert an p_{out}, der im vorhergehenden Schritt ausgegeben wurde: $\forall t : \ p(t+1) = p_{out}(t) \quad \text{und} \quad p(0) = \varepsilon$.
- Ist p frei (d. h., mit keinem Ausgabeport verbunden, keinem Eingabeport der Oberkom-

ponentenschnittstelle zugewiesen, und nicht Teil der externen Systemschnittstelle), so ist er stets leer: $\forall t : p(t) = \varepsilon$.

Für einen Ausgabeport p ist die Fallunterscheidung noch einfacher; da die Werte an Ausgabeports der Teilkomponenten durch ihre eigenen Verhaltensspezifikationen definiert sind, müssen nur die Ausgabeports der zusammengesetzten Komponente betrachtet werden:

- Ist p einem Ausgabeport p_{out} einer Teilkomponente C_i zugewiesen, so ist sein Wert gleich dem Wert an p_{out}: $\forall t : p(t) = p_{out}(t)$.
- Ist p frei so ist er stets leer: $\forall t : p(t) = \varepsilon$.

Eine vollständige formale Semantikdefinition für zusammengesetzte Komponenten findet sich in [Tra09b, Abschnitt 4.1.2].

4 Mehrtaktsemantik – Modulare Zeitverfeinerung

Die uniforme Taktung aller Komponenten bei der Eintaktsemantik hat, neben dem Vorzug eines sehr einfachen Ausführungsmodells, drei wesentliche Nachteile:

- Struktur/Komposition: Ist in einer zusammengesetzten Komponente die Länge eines Verarbeitungspfads gleich k, so kann sie das Berechnungsergebnis für eine Eingabe über diesen Pfad frühestens nach k Schritten liefern (Abb. 3). Die Struktur einer Komponente beeinflusst damit die logische Verarbeitungszeit, was die strukturelle Modularität einschränkt: eine Änderung der Struktur, z. B. im Zuge der Systementwicklung, kann dann zu einer unbeabsichtigten (und unerwünschten) Änderung der Verarbeitungsdauer führen.
- Verhalten/Automaten: Durch die Gleichheit des Kommunikationstakts und des (internen) Ausführungstakts muss für eine Eingabe nach einem Zustandswechsel bereits die Ausgabe berechnet sein – alle Berechnungen und Fallunterscheidungen müssen in einer Transition codiert werden. Dadurch wird die Berechnungsstrukturierung eingeschränkt, die durch Verwendung von Zustandsautomaten erreicht werden soll.
- Reale Systeme: Reale eingebettete Systeme können aus mehreren Komponenten/Tasks auf mehreren Steuergeräten bestehen, die über Kommunikationsbusse verbunden sind (Abb. 7). Dadurch können sich die Geschwindigkeiten der Ausführung der einzelnen Tasks, der Kommunikation zwischen Tasks in einem Steuergerät, und der Kommunikation zwischen Tasks auf verschiedenen Steuergeräten um Größenordnungen unterscheiden. Insbesondere ist der interne Ausführungstakt wesentlich höher als der Kommunikationstakt (einige Mikrosekunden für einen Berechnungsschritt vs. einige Millisekunden für Nachrichtenübertragung über einen Bus). Damit bildet die uniforme Taktung eines Modells die reale Zielplattform unzureichend ab.

Die beschriebenen Nachteile beseitigen wir nun durch Erweiterung der AUTOFOCUS-Semantik um die *Mehrtaktsemantik*. Diese ermöglicht die modulare Zeitverfeinerung einer Komponente, indem ihr interner Ausführungstakt ein Vielfaches des Kommunikationstakts betragen darf.

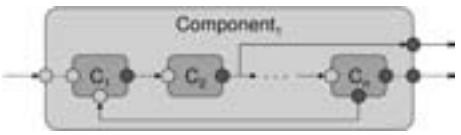


Abbildung 3: Gleichtakt: einfacher interner Ausführungstakt

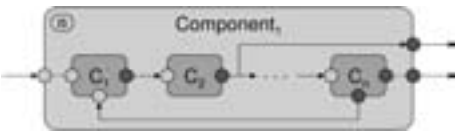


Abbildung 4: Zeitverfeinerung: mehrfacher interner Ausführungstakt

Eine Komponente mit Taktfaktor $k \in \mathbb{N}_+$ hat k Schritte zur Verarbeitung einer Eingabe. Die herkömmliche Eintaktsemantik ist nun Spezialfall der Mehrtaktsemantik für $k = 1$. Für $k > 1$ wird die Eingabe und Ausgabe einer zeitverfeinerten Komponente im internen Ausführungszyklus von k Schritten (entsprechend einem externen Kommunikationszyklus) wie folgt verarbeitet. Seien p_{in} und p_{out} Eingabe- bzw. Ausgabeports und $t \in \mathbb{N}$.

- Die intern sichtbare Eingabe zu Beginn des t-ten Ausführungszyklus der Länge k ist gleich der extern empfangenen Eingabe und ist leer in allen anderen Schritten dieses Ausführungszyklus:
$$p_{in}^{(int)}(t * k + i) = \begin{cases} p_{in}^{(ext)}(t) & \text{falls } i = 0 \\ \varepsilon & \text{falls } i \in \{1, \ldots, k-1\} \end{cases}$$
- Die extern sichtbare Ausgabe am Ende des Ausführungszyklus ist gleich der letzten nicht-leeren Nachricht, die im Laufe dieses Zyklus intern ausgegeben wurde, bzw. leer, falls alle internen Ausgaben leer waren (entspricht dem Verhalten eines Puffers der Größe 1, der vor jedem Ausführungszyklus geleert wird):
$$p_{out}^{(ext)}(t) = \begin{cases} \varepsilon & \text{falls } \forall i \in \{0, \ldots, k-1\} : p_{out}^{(int)}(t * k + i) = \varepsilon \\ p_{out}^{(int)}(t * k + i) & \text{falls } i = \mathsf{max}\{\, i \in \{0, \ldots, k-1\} \mid p_{out}^{(int)}(t * k + i) \neq \varepsilon \,\} \end{cases}$$

Die Abb. 5 zeigt schematisch die Eingabe-/Ausgabeverarbeitung einer zeitverfeinerten Komponente mit Taktfaktor 3.

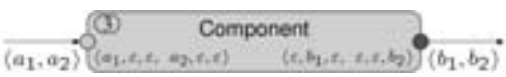


Abbildung 5: Mehrtaktsemantik: Eingabe und Ausgabe einer zeitverfeinerten Komponente

Auf diese Weise wird die modulare Zeitverfeinerung durch Änderungen allein innerhalb der Kommunikationsschnittstelle von Komponenten ermöglicht – es sind keine weiteren Änderungen der Modellierungstechnik notwendig. Zudem sind auch keine extern sichtbaren Änderungen für Kommunikationspartner zu berücksichtigen. Die einzige Änderung ist, dass einer Komponente für die Verarbeitung einer jeden Eingabe intern k Schritte zur Verfügung stehen, weil auf eine nicht-leere Eingabe stets $k - 1$ leere Eingaben folgen.

In [Tra09b, Abschnitt 4.1.2] wird die Mehrtaktsemantik formal definiert. Ferner wird gezeigt, wie eine zeitverfeinerte Komponente mit Taktfaktor k in den externen Kommunikationstakt durch die Expansion ihrer Eingabeströme und Kompression ihrer Ausgabeströme um den Faktor k eingebunden wird, und der Beweis geführt, dass diese Einbindung zu der oben beschriebenen Mehrfachtaktung einer zeitverfeinerten Komponente äquivalent ist.

5 Eigenschaften der Mehrtaktsemantik

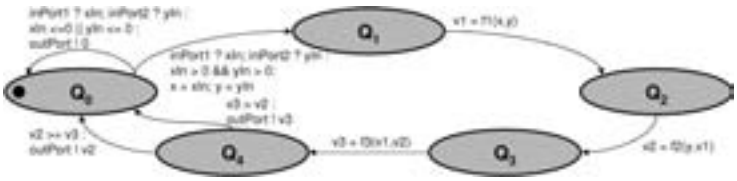


Abbildung 6: Automat mit strukturierter Berechung und Leerlaufzustand

Wir wollen nun auf die Auswirkungen der Mehrtaktsemantik eingehen. In Abschnitt 4 wurden die durch uniforme Taktung aller Komponenten entstehenden Einschränkungen erörtert. Diese können durch modulare Zeitverfeinerung behoben werden:

- Struktur/Komposition: Für eine zusammengesetzte Komponente kann der Taktfaktor k so gewählt werden, dass die Verarbeitung der Eingabe auch auf dem längsten in der Komponente vorhandenen Verarbeitungspfad innerhalb eines internen Ausführungszyklus der Länge k abgeschlossen werden kann. So werden in der Abb. 4 durch die Wahl des Taktfaktors n für jede Eingabe n interne Takte und damit genau 1 (externer) Kommunikationstakt benötigt. Dank der Puffersemantik der Ausgabeports dürfen Ausgaben zu verschiedenen Zeitpunkten des internen Ausführungszyklus erfolgen. Außerdem können dank der starken Kausalität der Kommunikationssemantik auch Feedback-Schleifen ohne Einschränkungen verwendet werden.
- Verhalten/Automaten: Für ein Zustandsübergangsdiagramm ist es nun möglich, für eine Eingabe mehrere Transitionen auszuführen, so dass die Berechnung durch Verwendung mehrerer Kontrollzustände und Transitionen besser strukturiert werden kann. So könnte zwar die Berechnung in der Abb. 6 auch in einer Transition codiert werden, diese müsste jedoch mehrere Berechnungsschritte und Fallunterscheidungen zu einem (großen und unübersichtlichen) Ausdruck zusammenfassen.[1]

  Ferner können durch eine einfache syntaktische Analyse die *Leerlaufzustände* eines Automaten ermittelt werden – in diesen Zuständen kann keine Transition schalten und erfolgt auch keine Ausgabe, bis eine nicht-leere Eingabe eingeht (in Abb. 6 ist dies der Zustand Q_0). Ähnlich wie bei zusammengesetzten Komponenten oben, genügt hier die Wahl eines hinreichend hohen Taktfaktors k, so dass für jede Eingabe nach spätestens k Schritten ein Leerlaufzustand erreicht wird – in diesem Fall wird die Komponente das Berechnungsergebnis für jede Eingabe am Ende des Ausführungszyklus der Länge k und somit vor Beginn des nächsten Kommunikationstakts liefern.
- Reale Systeme: Die Mehrtaktsemantik ermöglicht eine bessere Abbildung der Architektur realer eingebetteter Systeme durch AUTOFOCUS-Modelle. Hierfür können Komponenten, die Steuergeräte darstellen und Tasks als Teilkomponenten beinhalten, mit verschiedenen internen Taktfaktoren betrieben werden – das bildet zum einen die

[1] Folgender Ausdruck beschreibt beispielsweise die Ausgabe dieses Automaten am Port `outPort`:
`if (x <= 0 || y <= 0) then 0 else let v1 = f1(x, y) in let v2 = f2(y, v1) in let v3 = f3(v1, v2) in if (v3 > v2) then v3 else v2 fi tel tel tel fi`.

Tatsache ab, dass reale Steuergeräte unterschiedlich schnell sein können, und zum anderen, dass ein einzelner Rechenschritt in einem Steuergerät erheblich weniger Zeit benötigt, als die Nachrichtenübermittlung durch einen Kommunikationsbus. Für Komponenten, die Tasks darstellen, kann durch unterschiedliche Taktfaktoren die unterschiedliche Aufteilung der Rechenzeit auf die Tasks eines Steuergeräts modelliert werden. So zeigt die Abb. 8 schematisch, wie ein AUTOFOCUS-Modell des Systems in der Abb. 7 aussehen könnte (der Bus muss hierbei nicht explizit modelliert werden, da Kommunikationskanäle eine Abstraktion des Kommunikationsmechanismus darstellen und eine separate Buskomponente überflüssig machen).

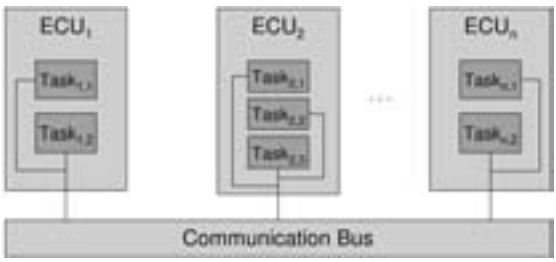


Abbildung 7: Architektur realer verteilter eingebetteter Systeme

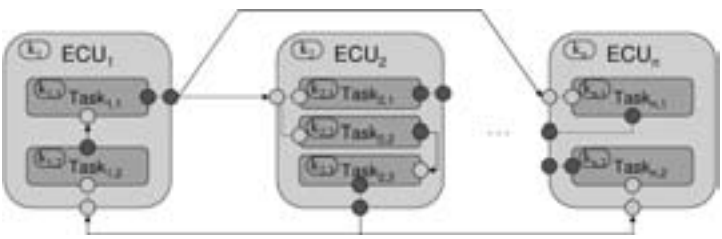


Abbildung 8: Zeitverfeinerung: Modellierung der Architektur realer Systeme

6 Verwandte Arbeiten

Wir betrachten nun einige Ansätze, die in verschiedenen modellbasierten Entwicklungstechniken/Werkzeugen zur Modellierung von unterschiedlichen Ausführungsgeschwindigkeiten sowie von Berechnungen, die mehrere einzelne Berechnungs-/Verarbeitungsschritte benötigen, verwendet werden.

FOCUS-Zeitverfeinerung Ein eng verwandter Ansatz ist die Zeitverfeinerung in FOCUS [Bro09]. Hier werden Operatoren zur Vergröberung und Verfeinerung der zeitlichen Auflösung von Nachrichtenströmen verwendet. Ein wesentlicher Unterschied zur AUTOFOCUS-Mehrtaktsemantik ist, dass Ströme in FOCUS mehrere Nachrichten pro Zeiteinheit enthalten dürfen – daher können bei Zeitvergröberung um einen Faktor $k \in \mathbb{N}$ alle Nach-

richten aus k Zeiteinheiten zu einer Nachrichtensequenz konkateniert werden, während in AUTOFOCUS, wo nur eine Nachricht pro Zeiteinheit zugelassen ist, alle k Nachrichten zu einer Nachricht aggregiert werden müssen.

Strukturelle Modularität/Verzögerungskumulation auf Verarbeitungspfaden
In zahlreichen Modellierungstechniken/-Werkzeugen wird eine verzögerungsfreie Kommunikationssemantik (Synchronie-Annahme, [CPHP87]), wie in Datenflussnetzwerken, verwendet. Der Vorteil dieser Technik ist, dass die Eingaben einer Komponente stets gleichzeitig ankommen, unabhängig davon, wie lang der Verarbeitungspfad jeder der Eingaben ist. Dies wird durch den Nachteil erkauft, dass beim Feedback von Ausgaben einer Komponente C_2 an eine Komponente C_1, deren Ausgabe wiederum als Eingabe von C_2 verwendet wird, die Kausalität möglicherweise nicht mehr gewahrt wird: das Verhalten solcher Netzwerke ist Gegenstand dedizierter Forschung [SBT96]. Eine praktikable Lösung ist auch explizites Einfügen von Verzögerungsgliedern, wodurch starke Kausalität induziert wird.

Da in der AUTOFOCUS-Mehrtaktsemantik einzelne Komponenten mit beliebigem festem internem Ausführungstakt betrieben werden können, kann den auf internen Verarbeitungspfaden entstehenden Verzögerungskumulationen durch die Wahl eines hinreichenden Taktfaktors begegnet werden, so dass das Berechnungsergebnis für Eingaben im Schritt t tatsächlich am Ende des Berechnungsschritts t ausgegeben wird und im nächsten Schritt $t + 1$ an verbundene Komponenten weitergeleitet wird, ohne dass eine strukturabhängige längere Verzögerung extern beobachtbar ist. Gleichzeitig wird durch die starke Kausalität der Kommunikationssemantik die uneingeschränkte Verwendung von Feedback-Schleifen ermöglicht.

Unterschiedliche Ausführungsgeschwindigkeit von Komponenten durch Clocks
In MATLAB/Simulink und den synchronen Sprachen wie Lustre können Multirate-Systeme durch explizite Taktgeber bzw. Uhren mit Unterabtastung modelliert werden, d.h., ein Block/Komponente wird nicht in jedem Schritt, sondern nur zu manchen Zeitpunkten bezüglich einer globalen Uhr aktiviert. In diesem Fall kann die Modellierung der Zeitverfeinerung unübersichtlich werden, da alle Uhren explizit über die globale Uhr zusammenhängen. SIGNAL erlaubt auch Überabtastung, indem Uhren schneller als eine globale Uhr laufen können (Polychronie) [GTL03]: dieser Ansatz ist am ehesten mit der Mehrtaktsemantik verwandt. Jedoch wird wegen der Flexibilität, verschiedenen Signalen verschiedene Uhren zuweisen zu können, ein eigener Clock-Kalkül benötigt. Die Mehrtaktsemantik ist hingegen pragmatischer und intuitiver zu handhaben: zur Zeitverfeinerung einer Teilkomponente C_i einer Komponente C genügt die Angabe des Taktfaktors $k \in \mathbb{N}_+$, um den C_i relativ zur Taktung von C schneller ist.

Strukturierung von Automatenberechnungen Auf Statecharts [Har87] basierende Techniken, wie beispielsweise MATLAB/Stateflow oder IBM Rational Rhapsody ermöglichen die Durchführung mehrerer Berechnungsschritte für eine Eingabe – die Zustandstransitionen werden durchgeführt, bis kein transitionsauslösendes Ereignis mehr erzeugt wird (Run-To-Completion) [HK04]. Dadurch können in einem Berechnungsschritt Endlosschleifen und damit nicht-terminierende Berechnungen auftreten. Die Behandlung dieses Problems ist ebenfalls Gegenstand dedizierter Forschung: so wird in [SSC+04] für die Übersetzung in Lustre eine "sichere" Teilsprache von Stateflow definiert.

Die Mehrtaktsemantik ermöglicht durch Einstellung eines festen Taktfaktors $k \in \mathbb{N}_+$ längere Berechnungen eines Zustandsautomaten – die Berechnung endet stets innerhalb eines Kommunikationstakts, wenn sie nicht mehr als k Zustandswechsel benötigt. Hierdurch wird die Run-To-Completion-Semantik für Berechnungen bis zu einer festen Maximallänge ermöglicht, und gleichzeitig die Terminierung jeder einzelnen Berechnung durch den festen Taktfaktor garantiert. Das Konzept der Leerlaufzustände erlaubt zudem eine effiziente syntaktische Analyse zur Ermittlung von Leerlaufzuständen, bei deren Erreichen eine Berechnung stets terminiert, und der Automat im Leerlauf verbleibt, bis eine neue nichtleere Eingabe ankommt.

7 Fazit

In diesem Artikel stellten wir eine wirksame und gleichzeitig einfache und robuste Erweiterung des AUTOFOCUS-Ausführungssemantik um die modulare Zeitverfeinerung einzelner Komponenten durch Mehrfachtaktung vor. Die modulare Zeitverfeinerung ermöglicht durch kontrollierte Entkopplung des Ausführungstakts vom Kommunikationstakt eine flexiblere Modellierung von Berechnungen und Komponentenstrukturen. Sie befördert dabei insbesondere die Modularität der Architekturentwürfe, denn nun können Komponenten ohne Seiteneffekte durch andere Komponenten ersetzt werden, die intern zwar eine andere Struktur und Verarbeitungsdauer haben dürfen, jedoch extern durch Zeitverfeinerung das gleiche Verhalten präsentieren. Damit wird ein wesentliches Prinzip modularer Systementwicklung für die AUTOFOCUS-Modellierungstechnik realisiert.

Zu den zukünftigen Weiterentwicklungen gehört vor allem die praktische Implementierung modularer Zeitverfeinerung durch Mehrtaktsemantik im AUTOFOCUS-Werkzeug. Zum anderen ist die Entwicklung von Codegeneratoren für verschiedene Zielwerkzeuge (Implementierungssprachen C und Java, aber auch Verifikationswerkzeuge wie SMV und Isabelle/HOL), wie sie früher für die herkömmliche Eintaktsemantik durchgeführt wurde, eine wichtige praktische Richtung für zukünftige Arbeiten.

Literatur

[ABR05] A. Angermann, M. Beuschel und M. Rau. *Matlab – Simulink – Stateflow*. Oldenbourg, 2005.

[ASC] The ASCET Product Family. `http://www.etas.com/en/products/ascet_software_products.php`.

[Bro09] M. Broy. Relating Time and Causality in Interactive Distributed Systems. In M. Broy, W. Sitou und C.A.R. Hoare, Hrsg., *Engineering Methods and Tools for Software Safety and Security*, Jgg. 22 of *NATO Science for Peace and Security Series - D: Information and Communication Security*. IOS Press, 2009.

[BS01] M. Broy und K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs und J. Plaice. LUSTRE: a Declarative Language for Programming Synchronous Systems. In *POPL '87*, Seiten 178–188. ACM Press, 1987.

[Fri03] H.-G. Frischkorn. IT im Automobil – Innovationsfeld der Zukunft. Invited Workshop Keynote, Automotive Software Engineering and Concepts, INFORMATIK 2003: 33. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2003.

[Gri03] K. Grimm. Software Technology in an Automotive Company – Major Challenges. In *ICSE 2003*, Seiten 498–505. IEEE Computer Society, 2003.

[GTL03] P. Le Guernic, J.-P. Talpin und J.-C. Le Lann. POLYCHRONY for System Design. *Journal of Circuits, Systems and Computers*, 12(3):261–304, 2003.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Her06] G. Hertel. Mercer-Studie Autoelektronik – Elektronik setzt die Impulse im Auto, 2006.

[HK04] D. Harel und H. Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML) – Preliminary Version. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder und E. Westkämper, Hrsg., *SoftSpez Final Report*, Jgg. 3147 of *LNCS*, Seiten 325–354. Springer, 2004.

[HSE97] F. Huber, B. Schätz und G. Einert. Consistent Graphical Specification of Distributed Systems. In John Fitzgerald, Cliff B. Jones und Peter Lucas, Hrsg., *FME '97*, Jgg. 1313 of *LNCS*, Seiten 122–141. Springer, 1997.

[HSSS96] F. Huber, B. Schätz, A. Schmidt und K. Spies. AutoFocus - A Tool for Distributed Systems Specification. In B. Jonsson und J. Parrow, Hrsg., *FTRTFT '96*, Jgg. 1135 of *LNCS*, Seiten 467–470. Springer, 1996.

[PBKS07] A. Pretschner, M. Broy, I. H. Krüger und T. Stauner. Software Engineering for Automotive Systems: A Roadmap. In *FOSE 2007: ICSE 2007, Future of Software Engineering*, Seiten 55–71. IEEE Computer Society, 2007.

[Rei06] M. Reinfrank. Why is automotive software so valuable?: or 5000 lines of code for a cup of gasoline less (Keynote Talk). In *SEAS 2006: ICSE Workshop on Software Engineering for Automotive Systems*, Seiten 3–4. ACM, 2006.

[SBT96] T. R. Shiple, G. Berry und H. Touati. Constructive Analysis of Cyclic Circuits. In *EDTC 1996*, Seite 328. IEEE Computer Society, 1996.

[SCA] SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite/`.

[SSC+04] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis und F. Maraninchi. Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre. In G. C. Buttazzo, Hrsg., *EMSOFT 2004*, Seiten 259–268. ACM, 2004.

[Tra09a] D. Trachtenherz. Ausführungssemantik von AutoFocus-Modellen: Isabelle/HOL-Formalisierung und Äquivalenzbeweis. Tech. Rep. TUM-I0903, TU München, Jan 2009.

[Tra09b] D. Trachtenherz. *Eigenschaftsorientierte Beschreibung der logischen Architektur eingebetteter Systeme*. Dissertation, TU München, 2009.

[vdB94] M. v. d. Beeck. A Comparison of Statecharts Variants. In H. Langmaack, W. P. de Roever und J. Vytopil, Hrsg., *FTRTFT '94*, Jgg. 863 of *LNCS*, Seiten 128–148. Springer, 1994.

Modeling and Verifying Dynamic Communication Structures based on Graph Transformations*

Stefan Henkler, Martin Hirsch, Claudia Priesterjahn, Wilhelm Schäfer

Abstract: Current and especially future software systems increasingly exhibit so-called self* properties (e. g. , self healing or self optimization). In essence, this means that software in such systems needs to be reconfigurable at runtime to remedy a detected failure or to adjust to a changing environment. Reconfiguration includes adding or deleting software components as well as adding or deleting component interaction. As a consequence, the state space of self* systems becomes so complex, that current verification approaches like model checking or theorem proving usually do not scale. Our approach addresses this problem by first defining a so-called "regular" system architecture with clearly defined interfaces and predefined patterns of communication such that dependencies between concurrently running component interactions are minimized with respect to the system under construction. The construction of such architectures and especially its reconfiguration is controlled by using graph transformation rules which define all possible reconfigurations. It is formally proven that such a rule set cannot produce any "non-regular" architecture. Then, the verification of safety and liveness properties has to be carried out for only an initially and precisely defined set of so-called coordination patterns rather than on the whole system.

1 Introduction

Current and especially future software systems increasingly exhibit so-called self* properties (e. g. , self healing or self optimization). In essence, this means that software in such systems needs to be reconfigurable at runtime to remedy a detected failure or to adjust to a changing environment. Reconfiguration includes adding or deleting software components as well as adding or deleting component interaction. This increases the complexity of the software significantly because the reconfiguration process also has to be controlled by software.

As many of these systems are used in a safety critical environment, high quality software is absolutely necessary; in particular, it must satisfy safety and liveness constraints. A common approach (at least in research and advanced industrial projects) to address these challenges is to verify as many constraints as possible at the model-level, i. e. , the software model is formally analyzed and code is automatically generated from the model.

*This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

However, the state space of self* systems becomes so complex, that current verification approaches like model checking or theorem proving usually do not scale [Alu08]. Even when using a compositional approach to verification and applying symbolic techniques, it is hardly possible to track all possible reachable system states because the possible dependencies between different, usually highly concurrent component interactions in a complex network of components are by far too many.

On a general level, our approach addresses this problem by first defining a regular system architecture with clearly defined interfaces and predefined patterns of communication such that dependencies between concurrently running component interactions are minimized with respect to the system under construction. The construction of such architectures and especially its reconfiguration is controlled by exploiting a formal definition of the construction process and by proving based on that formal definition that an architecture includes in fact only dependencies which are absolutely necessary. In most cases, this means that reconfiguration can only add components and corresponding component interactions, if that does not interfere with already existing interactions which have been proven to be correct already. Then, the verification of safety and liveness properties has to be carried out for only an initially and precisely defined set of so-called coordination patterns rather than on the whole system.

We clearly distinguish between the internal behavior of a component and the interaction behavior between components that is based on message passing. In this paper, we focus on modeling and analyzing the communication structure among different components, i. e. , the specification of communication protocols.

Self*-systems are usually built as a network of components, possibly even mobile, and exhibit a high degree of component interaction. Often, this interaction must satisfy strict real-time constraints while of course still guaranteeing the mentioned safety properties (e. g. , in the case of network failures). Because of this characteristic, the protocols built are often complex. An approach to model and verify such protocols is described in [GTB$^+$03]. However, it is restricted to the specification of bilateral communication based on (extended) timed automata and does not consider reconfiguration at runtime.

In this paper, we extend our so-called MECHATRONIC UML approach. (The name stems from the fact that the approach has originally been developed in the domain of mechatronic systems but it is, of course, not restricted to this class of applications. In fact, a significant part of the software of advanced mechatronic systems deals with component interaction and adjusting their behavior concerning e.g. self-healing or self-optimization. In that sense, mechatronic systems are just a special case of self*-systems [SW07].) We add graph transformation rules to the original approach, i.e. a generating system, to formally define the construction of component architectures, as mentioned above. This approach also supports the formal analysis of certain properties of the resulting (reconfigurable) architectures which in turn minimizes the verification effort to check safety and liveness properties of the communication protocols.

The key contribution of this paper is thus a particular modeling and verification approach based on a combination of graph transformation rules and timed automata. It addresses the challenge of scalability of verification even in cases where a system does not only exhibit an arbitraryly large finite number of states but an infinite number of states.

A concrete example for a complex self*-system with the need to coordinate a varying number of components is the RailCab project[1]. The vision of the RailCab project is a mechatronic rail system where autonomous vehicles called shuttles apply the linear drive technology, as used by the Transrapid system, but travel on the existing passive track system of a standard railway system. This system is currently under construction and a first version of the controlling software of the physically existing system has been built using the approach of [HTBS08].

One particular problem (previously presented in [GTB+03]) is the convoy coordination of certain system components, e.g. the shuttles. Shuttles drive in a convoy in order to reduce energy consumption caused by air resistance and to achieve a higher system throughput. Such convoys are established on-demand and require small distances between the shuttles. These required small distances cause the real-time coordination between the speed control units of the shuttles to be safety critical which results in a number of constraints, that have to be addressed when building the shuttles' control software. In addition a complex coordination is required when the convoy consists of more than two shuttles. Since shuttles can join or leave a convoy during runtime a flexible structure for the specification of the coordination is needed.

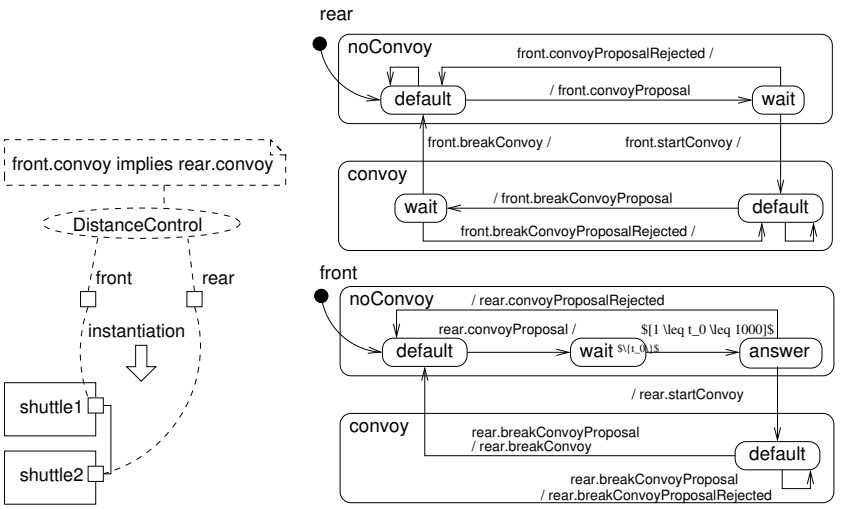
The structure of the paper is as follows: We first review the original MECHATRONIC UML approach in Section 2. Section 3 covers the concepts for modeling parameterized coordination patterns with a flexible number of participants and multi-ports. Then, we describe how compositional verification of our approach is achieved in Section 4. Finally, we will discuss related work in Section 5. The last section summarizes the paper and gives an outlook on future work.

2 Foundations

In our approach, the architecture is given by components, their ports and the connections between ports (which are just identified by links). This model is formally described by an adaptation of the UML 2.0 component model. Among other things, the adaptation especially covers the definition of restrictions on how ports have to be connected such that communication via different ports of the same component is guaranteed to be side-effect-free.

Communication between autonomous components has been defined by so-called *coordination patterns* [GTB+03]. A coordination pattern, as depicted in Figure 1(a), describes the communication between two components and consists of multiple communication partners, called *roles*. Roles are linked by a connector. The communication behavior

[1] http://www-nbp.upb.de



(a) Structure (b) Role behavior

Figure 1: Real-Time Coordination Pattern for a Shuttle Convoy

of a role is specified by a real-time statechart.

Real-time statecharts are an extension of UML state machines which support more powerful concepts for the specification of real-time behavior. They are semantically based on the timed automata formalism such that a formal analysis is possible using the model checker UPPAAL[2].

The behavior of the connector is described by another real-time statechart that, in addition to the transport of the messages, models the possible delay and the reliability of the channel, which are of crucial importance for many systems.

Safety constraints which have to hold (and are model checked) for these patterns are either a so-called pattern constraint or a role invariant which concerns a property of a single role only. A role invariant specifies a property that has to be satisfied by the communication partner. A pattern constraint specifies a property that has to be satisfied by all communication partners and connectors. Both types are defined in TCTL[3].

The role behavior is refined by ports that build the interfaces of our components, i.e. the ports implement the external behavior as specified by the role behavior. The refinement has to respect the role behavior (do not add possible behavior or block guaranteed behavior) and additionally has to respect the guaranteed behavior of the roles given by its invariants [GTB+03].

An additional statechart for synchronization is used to describe required dependencies between role behavior (as the component behavior is not necessarily only a parallel composition of the different role behavior). This allows for the strict separation of communication

[2] www.uppaal.com
[3] TCTL: Timed Computation Tree Logic

behavior and internal component behavior. We have described the specification and analysis of the synchronization behavior for bilateral communication in [GTB+03]. We have presented an automatic synthesis of the synchronization behavior in [HGH+09].

In our application example, the coordination between two shuttles is modeled by the DistanceControl pattern. It consists of two roles, the front role and the rear role and one connector that models the link between the two shuttles. The pattern specifies the protocols to coordinate two successive shuttles.

Initially, both roles are in state noConvoy::default, which means that they are not in a convoy. The rear role decides to propose building a convoy or not. After the decision to propose a convoy, a message is sent to the other shuttle resp. its front role. The front role decides to reject or to accept the proposal after max. 1000 msec. In the first case, both statecharts revert to the noConvoy::default state. In the second case, both roles switch to the convoy::default state.

Eventually, the rear shuttle decides to propose a break of the convoy and sends this proposal to the front shuttle. The front shuttle decides to reject or accept that proposal. In the first case, both shuttles remain in convoy-mode. In the second case, the front shuttle replies by an approval message, and both roles switch into their respective noConvoy::default states.

A safety requirement of the pattern is that no collision happens. The pattern constraint enforces the shuttle role to be in state Convoy while the coordinator role is also in state Convoy (shuttle.convoy implies coordinator.convoy).

3 Modeling Parameterized Coordination Patterns

So far, our approach allows constructing and verifying system structures with fixed bilateral communication only, i.e. reconfiguration is not supported. As self*-systems can be composed of thousands of components which might be replaced at runtime, it is not possible to specify and verify the system as a whole.

We use graph transformation systems to define the construction and reconfiguration of architectures consisting of components and corresponding component interactions. A so-called start graph defines an initial configuration and a set of rules defines all allowed configurations of components and component interactions (without specifying any behavioral part). Based on the start graph, one can formally prove that such a generating system does not produce any so-called forbidden graph structure. Such a graph structure would include a sub graph (corresponding to an architecture as a result of a reconfiguration operation) which incorporates component interactions where no predefined corresponding coordination pattern exists.

This means that the behavior of component interactions is again specified by real-time statecharts as explained in the previous section. However, in order to cover possible multilateral component interactions, they have to be extended by introducing parameters in the definition of statecharts as it will be explained in section 3.2.

3.1 System Structure Specification

The reconfiguration of an embedded or mechatronic system architecture defined by graph transformation rules, is often time critical. Especially the creation or deletion of a port object or component might involve some complex operations. Verification of time constraints of reconfiguration operations should consequently be verifiable on the model level as well. We enhance our specification approach by adding clock instances and invariants over these clocks to a graph transformation rule to specify the time which is needed for the corresponding reconfiguration operation.

Our formal definition of Timed Graph Transformation Systems [Hir08] enables the formal verification, i. e. reachability analysis to prove the correctness of our system structure concerning time constraints across several reconfiguration operations as well.

Figure 2(a) shows a graph transformation rule for joining an existing convoy. A graph transformation rule consists of a left hand side and a right hand side. The left hand side gives the system configuration in which the rule can be applied, the right hand side shows the result of the application. Here, left and right hand side are depicted in one single graph using the notation $++$ or $--$ for edges (and nodes) which are created or deleted during rule application. Since the reconfiguration happens within a certain time interval, we add clocks to our graph transformation rules and invariants over these clocks.

In our example rule in Figure 2(a) we add the clock t. The invariant $t < 5$ specifies that the creation of a link between two shuttles takes maximum 5 time units. By the attribute $<< last >>$ we specify that a new shuttle will join the convoy at the last position only.

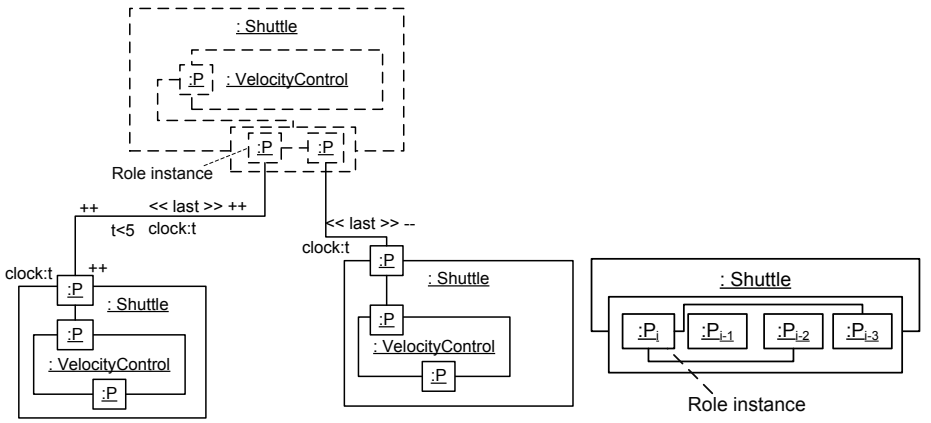
As graph transformation systems have a formal basis and tool support for model checking, a formal proof of properties is possible. In our case, properties which must not hold, would mean the generation of communication structures with no corresponding coordination pattern. We define such "non-regular" structures by so-called forbidden graphs.

Figure 2(b) depicts such a forbidden graph structure. It basically means that shuttles in a convoy or rather their member roles are not allowed to communicate with anything else than the next role instance in an ordering of role instances ($_i$, $_{i-1}$, ... reflects the ordering of the role instances).

3.2 Specification of Component Communication

To model the communication of more than two components we extend our coordination patterns to parameterized coordination patterns by introducing multi-roles. A multi-role consists of one or more sub-roles. The behavior of a multi-role is specified by a parameterized real-time statechart which defines the behavior of all sub-roles. Parameterized real-time statecharts are semantically based on a parameterized timed automaton, a timed automaton extended by parameterized signals.

Each sub-role of a multi-role is initialized by a coordination statechart (e. g. cf. Figure 5). A sub-role is created and initialized by a specific call, e. g. createPort, of a timed graph



(a) Extension Rule (b) Forbidden Graph Structure

Figure 2: Extension Rule and Forbidden Graph Structure

transformation rule. The formal specification of the sub-roles of a multi-role can be found in [Hir08].

If an additional synchronization statechart is required to describe dependencies between role behavior (see Section 2), we can apply our approach for bilateral communication [HHG08]. Here, we exploit the fact that the coordination statechart of a multi role encapsulates the set of role instances, while the internal synchronization uses the uniform interface.

In the following, we extend our example by the ConvoyCoordination pattern (see Figure 3), which enables the coordinator (role) to control the convoy (member role).

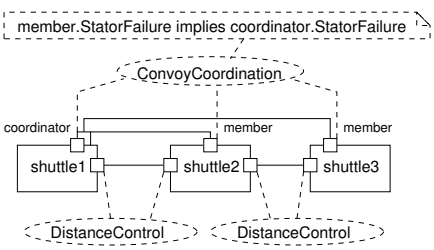


Figure 3: Extended convoy example

The parameterized role coordinator is depicted in Figure 4. Initially, the coordinator role is in state WaitForTrigger. If it receives a next_k trigger, the role switches to state Idle. Initially, next_k is sent by the adaption statechart as depicted in Figure 5 while in all following operations next_k is sent by the ancestor role with index $k-1$. Now, the communication with the shuttle role is started. First, an update message including current profile data, the reference velocity as well as the reference position of the shuttle, is sent. Thereafter, the role waits in state SentAcknowledge for the confirmation acknowledge by the shuttle

role. If this message is received the role switches to state WaitForTrigger and sends the next_{k+1} signal to activate the next sub-role. If the acknowledge message is not received, the role switches to state NextFailed and sends a signal to the internal component behavior in order to initiate appropriate routines. Further, the role switches to state StatorFailure, if a publishStatorFailure is received when being in state Idle.

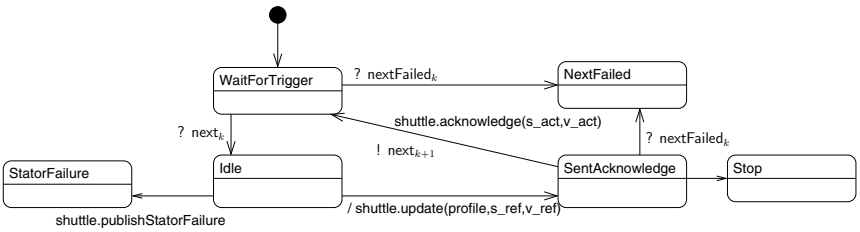


Figure 4: The behavior of a coordination role role_k

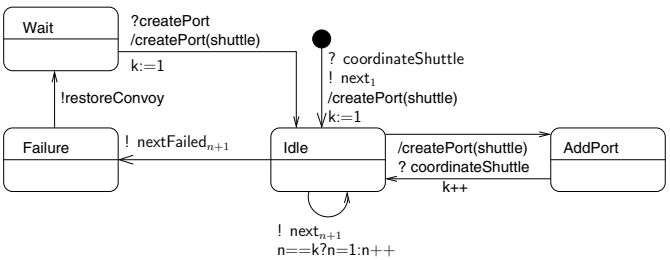


Figure 5: The coordination statechart for all coordinator roles

The real-time statechart of the member role consists of three states (cf. Figure 6). Initially, the role is in state Normal. Every 150 time units the role has to receive an update message from the coordinator role. This message includes the current profile, reference position and reference velocity. The role confirms the receipt of the message with an acknowledge message including the current position and current velocity. In case no update message is received (e.g. network failure) the role switches to the network failure state after 150 time units. The state StatorFailure will be reached if another shuttle propagates a StatorFailure.

The pattern constraint enforces the member role to be in state StatorFailure while the coordinator role is also in state StatorFailure (shuttle.StatorFailure implies coordinator.StatorFailure).

4 Verification

Figure 3 shows the communication structure of one coordinator and two shuttles. If we had created an arbitrary communication structure (by reconfiguration operations), we would have to verify the constraints for the whole state space which is computed by taking the concrete maximal instance number into account (which could be a very large number;

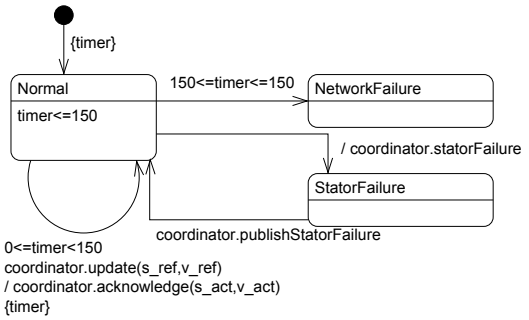


Figure 6: The behavior of a shuttle role member

maybe even infinite). By taking advantage of our regular system structure we only need to verify the constraints for an arbitrarily chosen coordination pattern and an arbitrarily selected pair of sub-roles.

The key idea to tackle these problems and to define a scalable verification approach is to exploit the underlying modeling formalism (cf. Section 3) and still use the compositional reasoning approach for bilateral communication [GTB+03].

The system achitecture is built such that component interactions do not interfere. This is achieved by the definition of forbidden graph structures as shown in Figure 2(b). Consequently, it is enough to verify an arbitrarily chosen pair of sub-roles concerning safety and liveness constraints[4]. Each sub-role corresponds to one port of a communication channel. If such a regular structure is correctly defined by a graph transformation system, it can be formally proven by induction over the number of sub-roles that the behavior of other sub-roles does not interfere with arbitrarily chosen pairs of sub-roles.

The induction is based on reachability analysis by the tool GROOVE [Ren04]. We have extended GROOVE by time to examine our timed graph transformation rules. In detail, all time constraints specified in the rules are considered and it is automatically checked if a specified invalid communication structure can be created (verification step 2).

To check the safety constraint shuttle.convoy implies coordinator.convoy as introduced in Section 2 for a pair of parametrized sub-roles we use the model checker UPPAAL[5] which we have successfully used in all our previous modeling and verification approaches, e. g. [BGHS04] (verification step 3). For completeness reasons we also have to check local invariant properties to ensure correct behavior of a single sub-role (verification step 1).

In the following we give a sketch of the necessary verification steps based on the definitions introduced in Section 3 to check if a parameterized coordination pattern is correct w. r. t. to the specified constraints, e. g. shuttle.convoy implies coordinator.convoy.

[4]This is correct for local constraints like safety constraints. For some global constraints the complete behavior of the system still has to be checked. For these constraints we refer to simulation. However, we did not find constraints of this type in our applications.

[5]www.uppaal.com

Verification steps:

1. For the parameterized coordination pattern verify that an arbitrary sub-role fulfills the local constraints of the multi-role, i.e. the roles are compatible, and there exists no deadlock.
2. Verify that the structural constraints of the timed graph transformation rules are satisfied, i.e. apply reachability analysis for timed graph transformation systems to verify that
   a) the timing constraints of the timed graph transformation rules are satisfied (see Figure 2(a)) and
   b) the situations specified by the forbidden graph structure do not occur (see Figure 2(b)).
3. Verify that pairs of subsequent sub-roles do not contain a deadlock.

5 Related Work

The state explosion problem limits the applicability of model checking for complex software systems. A number of modular and compositional verification approaches have therefore been proposed. One particular compositional approach is the assume/guarantee paradigm [MC81]. Model checking techniques that permit compositional verification following the assume/guarantee paradigm have been developed [CGP00, p. 185ff]. Our approach also employs the assume/guarantee paradigm, but supports dynamic structures and takes advantage of information available in form of communication structures and pattern role protocols to derive the required additional assumed and guaranteed properties automatically rather than manually as in [CGP00]. None of the current approaches for the specification and verification of dynamic systems considers a well-defined combination of the specification of a dynamic structure and its behavior to facilitate a compositional verification approach [BCDW04].

6 Conclusion and Future Work

Model-driven development of self*, mobile systems should support a formal verification approach, as many of these systems are used in a safety-critical environment. To a large extent, the behavior of such systems is given by the communication between the system components. We extend the existing model-driven MECHATRONIC UML approach to deal especially with the possibly infinite state space of self*, mobile systems. The infinite state space makes it basically impossible to apply existing "'classical"' approaches to system verification. The extensions of MECHATRONIC UML include the definition and verification of regular communication structures based on timed graph transformation

systems. As a consequence, only predefined communication protocols (coordination patterns) specifying bilateral as well as multilateral communication, need to be individually verified which is much less time consuming (and in fact possible for real applications) than checking a complete system specification. Component internal behavior in turn can be verified independently from the communication behavior but this is not the subject of this paper. Basically, the approach suggests a particular system decomposition enabling compositional verification.

Admittedly there is still one significant limit of the approach. Broadcasting based communication architectures where communication between components basically does not follow any rules (and does not exhibit corresponding hierarchies), cannot be covered by our approach. However, we found many examples of systems where this type of generality does not exist and thus our approach is applicable to a wide range of system like e. g. public transport, production or telecommunication systems. In fact, one could even argue that building "really" safe systems requires to avoid broadcasting like communication structures. Even the automobile industrie (where production costs are a major issue) works on approaches which refrain from broadcasting based communication architectures [NS08]. A first step to apply a structured modeling and analysis approach is realized by integrating the FlexRay[6] communication system. FlexRay requires a structured specification of the communication properties to enable deterministic communication.

A first evaluation of our approach was given in [HTBS08]. It describes the software development of the RailCab project (cf. section 1). As said, RailCab is not just a lab experiment. A real running prototype has been physically built on campus in the the scale of 1:2.5. The complete communication software of the shuttles has been implemented using the approach as presented in this paper.

We also work on a project which integrates the approach into an existing commercial tool called CAMeL-View. This tool is used in control engineering to specify single controllers. State-based communication between controllers is supposed to be modelled and verified by the MECHATRONIC UML approach [GBSO04].

Acknowledgements We thank Tobias Eckardt and Markus von Detten for their comments and proof-reading of this paper.

References

[Alu08] Rajeev Alur. Model Checking: From Tools to Theory. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 89–106. Springer Berlin / Heidelberg, 2008.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM.

[6] http://www.flexray.com/

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real-Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, pages 1–20, October 2004.

[CGP00] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, January 2000.

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pages 179–188. ACM Press, November 2004.

[GTB+03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[HGH+09] Stefan Henkler, Joel Greenyer, Martin Hirsch, Wilhelm Schäfer, Kathan Alhawash, Tobias Eckardt, Christian Heinzemann, Renate Löffler, Andreas Seibel, and Holger Giese. Synthesis of Timed Behavior From Scenarios in the Fujaba Real-Time Tool Suite. In *Proc. of the 31th International Conference on Software Engineering (ICSE), Vancouver, Canada*, May 2009.

[HHG08] Martin Hirsch, Stefan Henkler, and Holger Giese. Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML. In *Proc. of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08),Leipzig, Germany*, pages 33–40. ACM Press, May 2008.

[Hir08] Martin Hirsch. *Modell-basierte Verifikation von vernetzten mechatronischen Systemen*. PhD thesis, University of Paderborn, Paderborn, Germany, September 2008.

[HTBS08] Christian Henke, Matthias Tichy, Joachim Böcker, and Wilhelm Schäfer. Organization and Control of Autonomous Railway Convoys. In *Proceedings of the 9th International Symposium on Advanced Vehicle Control, Kobe, Japan*, October 2008.

[MC81] J. Misra and M. Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[NS08] Oliver Niggemann and Joachim Stroop. Models for model's sake: why explicit system models are also an end to themselves. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 561–570. ACM, 2008.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer Verlag, 2004.

[SW07] Wilhelm Schäfer and Heike Wehrheim. The Challenges of Building Advanced Mechatronic Systems. In *FOSE '07: 2007 Future of Software Engineering*, pages 72–84, Washington, DC, USA, 2007. IEEE Computer Society.

Virtuelle Trennung von Belangen (Präprozessor 2.0)

Christian Kästner, Sven Apel, Gunter Saake

{kaestner,saake}@iti.cs.uni-magdeburg.de, apel@uni-passau.de

Abstract: Bedingte Kompilierung mit Präprozessoren wie *cpp* ist ein einfaches, aber wirksames Mittel zur Implementierung von Variabilität in Softwareproduktlinien. Durch das Annotieren von Code-Fragmenten mit *#ifdef* und *#endif* können verschiedene Programmvarianten mit oder ohne diesen Fragmenten generiert werden. Obwohl Präprozessoren häufig in der Praxis verwendet werden, werden sie oft für ihre negativen Auswirkungen auf Codequalität und Wartbarkeit kritisiert. Im Gegensatz zu modularen Implementierungen, etwa mit Komponenten oder Aspekte, vernachlässigen Präprozessoren die Trennung von Belangen im Quelltext, sind anfällig für subtile Fehler und verschlechtern die Lesbarkeit des Quellcodes. Wir zeigen, wie einfache Werkzeugunterstützung diese Probleme adressieren und zum Teil beheben bzw. die Vorteile einer modularen Implementierung emulieren kann. Gleichzeitig zeigen wir Vorteile von Präprozessoren wie Einfachheit und Sprachunabhängigkeit auf.

1 Einleitung

Der C-Präprozessor *cpp* und ähnliche Werkzeuge[1] werden in der Praxis häufig verwendet, um Variabilität zu implementieren. Quelltextfragmente, die mit *#ifdef* und *#endif* annotiert werden, können später beim Übersetzungsvorgang ausgeschlossen werden. Durch verschiedene Übersetzungsoptionen oder Konfigurationsdateien können so verschiedene Programmvarianten, mit oder ohne diese Quelltextfragmente, erstellt werden.

Präprozessoranweisungen sind zum Implementieren von *Softwareproduktlinien* sehr gebräuchlich. Eine Softwareproduktlinie ist dabei eine Menge von verwandten Anwendungen in einer Domäne, die alle aus einer gemeinsamen Quelltextbasis generiert werden können [BCK98, BKPS04]. Ein Beispiel ist eine Produktlinie für Datenbanksysteme, aus der man Produktvarianten entsprechend des benötigten Szenarios generieren kann [RALS09, Sel08], etwa ein Datenbanksystem mit oder ohne Transaktionen, mit oder ohne Replikation, usw. Die einzelnen Produktvarianten werden durch Features (oder Merkmale) unterschieden, welche die Gemeinsamkeiten und Unterschiede in der Domäne beschreiben [K$^+$90, AK09] – im Datenbankbeispiel etwa Transaktionen oder Replikation.

[1] Ursprünglich wurde *cpp* für Metaprogrammierung entworfen. Von seinen drei Funktionen (a) Einfügen von Dateiinhalten *(#include)*, (b) Makros *(#define)* und (c) bedingte Kompilierung *(#ifdef)* ist hier nur die bedingte Kompilierung relevant, die üblicherweise zur Implementierung von Variabilität verwendet wird. Neben *cpp* gibt es viele weitere Präprozessoren mit ähnlicher Funktionsweise. Zum Beispiel wird für Java-ME-Anwendungen der Präprozessor *Antenna* häufig verwendet, die Entwickler von Java's Swing Bibliothek implementierten einen eigenen Präprozessor *Munge*, die Programmiersprachen Fortran und Erlang haben ihren eigenen Präprozessor, und bedingte Kompilierung ist Bestandteil von Sprachen wie C#, Visual Basic, D, PL/SQL und Adobe Flex.

Eine Produktvariante wird durch eine Feature-Auswahl spezifiziert, z. B. „Die Datenbankvariante mit Transaktionen, aber ohne Replikation und Flash“. Kommerzielle Produktlinienwerkzeuge, etwa jene von *pure-systems* und *BigLever*, unterstützen Präprozessoren explizit.

Obwohl Präprozessoren in der Praxis sehr gebräuchlich sind, gibt es erhebliche Bedenken gegen ihren Einsatz. In der Literatur werden Präprozessoren sehr kritisch betrachtet. Eine Vielzahl von Studien zeigt dabei den negativen Einfluss der Präprozessornutzung auf Codequalität und Wartbarkeit, u.a. [SC92, KS94, Fav97, EBN02]. Präprozessoranweisungen wie *#ifdef* stehen dem fundamentalen Konzept der Trennung von Belangen entgegen und sind sehr anfällig für Fehler. Viele Forscher empfehlen daher, die Nutzung von Präprozessoren einzuschränken oder komplett abzuschaffen, und Produktlinien stattdessen mit ‚modernen‘ Implementierungsansätzen wie Komponenten und Frameworks [BKPS04], Feature-Modulen [Pre97], Aspekten [K+97] oder anderen zu implementieren, welche den Quelltext eines Features modularisieren.

Trotz allem stellen wir uns in diesem Beitrag auf die Seite der Präprozessoren und zeigen, wie man diese verbessern kann. Bereits einfache Erweiterungen an Konzepten und Werkzeugen und ein diziplinliertes Vorgehen können viele Fallstricke beseitigen. Daneben darf auch nicht vergessen werden, dass auch Präprozessoren neben den genannten Schwächen diverse Vorteile für die Produktlinienentwicklung besitzen. Wir zeigen, wie eigene und fremde Fortschritte an verschiedenen Fronten zu einer gemeinsamen Vision der virtuellen Trennung von Belangen zusammenwirken. Die Bezeichnung „Virtuelle Trennung von Belangen“ ergibt sich aus einer Erweiterung, die eine Trennung von Belangen emuliert, ohne den Quelltext wirklich in physisch getrennte Module zu teilen.

Schlussendlich muss noch erwähnt werden, dass auch die vorgestellten Erweiterungen nicht das letzte Wort zum Thema Produktlinienimplementierung sind. In unseren Arbeiten untersuchen wir sowohl modulare als auch präprozessorbasierte Implementierungsmöglichkeiten. Das Ziel dieses Beitrags ist, die in der Forschung unterrepräsentierten Präprozessoren und ihr noch nicht ausgeschöpftes Potential ins Bewusstsein der Forschergemeinde zu rücken.

2 Kritik an Präprozessoren

Im Folgenden werden die drei häufigsten Argumente gegen Präprozessoren vorgestellt: unzureichende Trennung von Belangen, Fehleranfälligkeit und unlesbarer Quelltext.

Trennung von Belangen. Die unzureichende Trennung von Belangen und die verwandten Probleme fehlender Modularität und erschwerter Auffindbarkeit von Quelltext eines Features sind in der Regel die größten Kritikpunkte an Präprozessoren. Anstatt den Quelltext eines Features in einem Modul (oder eine Datei, eine Klasse, ein Package, o.ä.) zusammenzufassen, ist Feature-relevanter Code in präprozessorbasierten Implementierungen in der gesamten Codebasis verstreut und mit dem Basisquelltext sowie dem Quelltext anderer Features vermischt. Im Datenbankbeispiel wäre etwa der gesamte Transaktionsquelltext (z. B. Halten und Freigabe von Sperren) über die gesamte Codebasis der Datenbank verteilt und vermischt mit dem Quelltext für Replikation und andere Features.

Die mangelnde Trennung von Belangen wird für eine Vielzahl von Problemen verantwortlich gemacht. Um das Verhalten eines Features zu verstehen, ist es zunächst nötig, den entsprechenden Quelltext zu finden. Dies bedeutet, dass die gesamte Codebasis durchsucht werden muss; es reicht nicht, ein einzelnes Modul zu durchsuchen. Man kann einem Feature nicht direkt zu seiner Implementierung folgen. Vermischter Quelltext lenkt zudem beim Verstehen des Quelltextes ab, weil man sich ständig mit Quelltext beschäftigen muss, der für die aktuelle Aufgabe nicht relevant ist. Die erschwerte Verständlichkeit des Quelltextes durch Verteilung und Vermischung des Quelltextes erhöht somit die Wartungskosten und widerspricht jahrzehntelanger Erfahrung im Software-Engineering.

Fehleranfälligkeit. Wenn Präprozessoren zur Implementierung von optionalen Features benutzt werden, können dabei sehr leicht subtile Fehler auftreten, die sehr schwer zu finden sind. Das beginnt schon mit einfachen Syntaxfehlern, da Präprozessoren wie *cpp* auf Basis von Textzeilen arbeiten, ohne den zugrundeliegenden Quelltext zu verstehen. Damit ist es ein leichtes, etwa nur eine öffnende Klammer, aber nicht die entsprechende schließende Klammer zu annotieren, wie in Abbildung 1 illustriert (die Klammer in Zeile 4 wird nur in Zeile 17 geschlossen, wenn das Feature *HAVE_QUEUE* ausgewählt ist; falls Feature *HAVE_QUEUE* nicht ausgewählt ist, fehlt dem resultierendem Program eine schließende Klammer). In diesem Fall haben wir den Fehler zu Anschauungszwecken selber eingebaut, aber ähnliche Fehler können leicht auftreten und sind schwer zu erkennen (wie uns mehrfach ausdrücklich von verschiedenen Produktlinienentwicklern bestätigt wurde). Die Verteilung des Featurecodes macht das Problem noch schwieriger.

Das größte Problem ist jedoch, dass ein Compiler solche Probleme bei der Entwicklung nicht erkennen kann, solange nicht der Entwickler (oder ein Kunde) irgendwann eine Produktvariante mit einer problematischen Featurekombination erstellt und übersetzt. Da es aber in einer Produktlinie sehr viele Produktvarianten geben kann (2^n für n unabhängige, optionale Features; industrielle Produktlinien haben hunderte bis tausende Features, beispielsweise hat der Linux Kernel über 8000 Konfigurationsoptionen [TSSPL09]), ist es unrealistisch, bei der Entwicklung immer alle Produktvarianten zu prüfen. Somit können selbst einfache Syntaxfehler, die sich hinter bestimmten Featurekombinationen verstecken, über lange Zeit unentdeckt bleiben und im Nachhinein (wenn ein bestimmtes Produkt generiert werden soll) hohe Wartungskosten verursachen.

Syntaxfehler sind eine einfache Kategorie von Fehlern. Darüber hinaus können natürlich genauso auch Typfehler und Verhaltensfehler auftreten, im schlimmsten Fall wieder nur in wenigen spezifischen Featurekombinationen. Beispielsweise muss beachtet werden, in welchem Kontext eine annotierte Methode aufgerufen wird. In Abbildung 2 ist die Methode *set* so annotiert, dass sie nur enthalten ist, wenn das Feature *Write* ausgewählt ist; ist es dagegen nicht ausgewählt, wird die Methode entfernt und es kommt zu einem Typfehler in Zeile 3, wo die Methode dennoch aufgerufen wird. Obwohl Compiler in statisch getypten Sprachen solche Fehler erkennen können, würde so ein Fehler wieder nur erkannt, wenn die problematische Featurekombination kompiliert wird.

```
1  static int __rep_queue_filedone(
       dbenv, rep, rfp)
2    DB_ENV *dbenv;
3    REP *rep;
4    __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6    COMPQUIET(rep, NULL);
7    COMPQUIET(rfp, NULL);
8    return (__db_no_queue_am(dbenv));
9  #else
10   db_pgno_t first, last;
11   u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
16   // weitere 100 Zeilen C Code
17 }
18 #endif
```

Abbildung 1: Modifizierter Quelltextauszug aus Oracle's Berkeley DB mit Syntaxfehler, wenn *HAVE_QUEUE* nicht ausgewählt ist.

```
1  class Database {
2    Storage storage;
3    void insert(Object key, Object data) {
4      storage.set(key, data);
5    }
6  }
7  class Storage {
8  #ifdef WRITE
9    boolean set(Object key, Object data) {
10     ...
11   }
12 #endif
13 }
```

Abbildung 2: Quelltextauszug mit Typfehler, wenn *WRITE* nicht ausgewählt ist.

Unlesbarer Quelltext. Beim Implementieren von Features mit *cpp* und ähnlichen Präprozessoren wird nicht nur der Quelltext verschiedener Features vermischt, sondern auch die Präprozessoranweisungen mit den Anweisungen der eigentlichen Programmiersprache. Beim Lesen des entsprechenden Quelltexts können eine Vielzahl von Präprozessoranweisungen vom eigentlichen Quelltext ablenken und zudem das gesamte Quelltextlayout zerstören (*cpp* erfordert, dass jede Anweisung in einer eigenen Zeile steht). Es gibt viele Beispiele, in denen Präprozessoranweisungen den Quelltext komplett zerstückeln – wie in Abbildung 3 gezeigt – und damit die Lesbarkeit und Wartbarkeit einschränken.

In Abbildung 3 wird der Präprozessor feingranular eingesetzt, um nicht nur Statements, sondern auch Parameter oder Teile von Ausdrücken zu annotieren [KAK08]. Durch Präprozessoranweisungen und zusätzliche notwendige Zeilenumbrüche werden insgesamt 21 statt neun Zeilen benötigt. Über das einfache Beispiel hinaus sind auch lange und geschachtelte Präprozessoranweisungen (siehe Abbildung 1) mitverantwortlich für schlechte Lesbarkeit.

Auch wenn das Beispiel in Abbildung 3 konstruiert wirkt, findet man ähnliche Beispiele in der Praxis. In Abbildung 4 sieht man etwa den Anteil an Präprozessoranweisungen im Quelltext des Echtzeitbetriebssystems *Femto OS*.

3 Virtuelle Trennung von Belangen

Nach einem Überblick über die wichtigsten Kritikpunkte von Präprozessoren diskutieren wir Lösungsansätze, die wir in ihrer Gesamtheit virtuelle Trennung von Belangen nennen. Diese Ansätze lösen nicht alle Probleme, können diese aber meist abschwächen. Zusammen mit den Vorteilen der Präprozessornutzung, die anschließend diskutiert wird, halten wir Präprozessoren für eine echte Alternative für Variabilitätsimplementierung.

```
1  class Stack {
2    void push(Object o
3  #ifdef TXN
4    , Transaction txn
5  #endif
6    ) {
7      if (o==null
8  #ifdef TXN
9        || txn==null
10 #endif
11       ) return;
12 #ifdef TXN
13     Lock l=txn.lock(o);
14 #endif
15     elementData[size++] = o;
16 #ifdef TXN
17     l.unlock();
18 #endif
19     fireStackChanged();
20   }
21 }
```

Abbildung 3: Java Quelltext zerstückelt durch feingranulare Annotationen mit *cpp*.

Abbildung 4: Präprozessoranweisungen in Femto OS (rote Linie = Präprozessoranweisung, weisse Linien = C-Code).

Trennung von Belangen. Eine der wichtigsten Motivationen für die Trennung von Belangen ist Auffindbarkeit, so dass ein Entwickler den gesamten Quelltext eines Features an einer einzigen Stelle finden und verstehen kann, ohne von anderen Quelltextfragmenten abgelenkt zu sein. Eine verteilte und vermischte präprozessorbasierte Implementierung kann dies nicht leisten, aber die Kernfrage „welcher Quelltext gehört zu diesem Feature“ kann mit *Sichten* trotzdem beantwortet werden [JDV04, SGC07, HKW08, KTA08].

Mit verhältnismäßig einfachen Werkzeugen ist es möglich, (editierbare) Sichten auf Quelltext zu erzeugen, die den Quelltext aller irrelevanten Features ausblenden. Technisch kann das analog zum Einklappen von Quelltext in modernen Entwicklungsumgebungen wie Eclipse implementiert werden.[2] Abbildung 5 zeigt beispielhaft ein Quelltextfragment und eine Sicht auf das darin enthaltende Feature *TXN*. Im Beispiel wird offensichtlich, dass es nicht ausreicht, nur den Quelltext zwischen *#ifdef*-Anweisungen zu zeigen, sondern dass auch ein entsprechender Kontext erhalten bleiben muss (z. B. in welcher Klasse und welcher Methode ist der Quelltext zu finden). In Abbildung 5 werden diese Kontextinformationen grau und kursiv dargestellt. Interessanterweise sind diese Kontextinformationen ähnlich zu Angaben, die auch bei modularen Implementierungen wiederholt werden müssen; dort be-

[2] Obwohl editierbare Sichten schwieriger zu implementieren sind als nicht-editierbare, sind editierbare Sichten nützlicher, da sie es dem Benutzer erlauben, den Quelltext zu ändern, ohne erst zurück zum Originalquelltext wechseln zu müssen. Lösungen für editierbare Sichten sind sowohl aus dem Bereich der Datenbanken wie auch aus bidirektionalen Modelltransformationen bekannt. Eine einfache, aber effektive Lösung, die auch in unseren Werkzeugen benutzt wird, ist es, Markierungen in der Sicht zu belassen, die ausgeblendeten Quelltext anzeigen. Änderungen am Quelltext vor oder nach der Markierung können so eindeutig in den Originalquelltext zurückpropagiert werden.

```
1  class Stack implements IStack {
2    void push(Object o) {
3  #ifdef TXN
4      Lock l = lock(o);
5  #endif
6  #ifdef UNDO
7      last = elementData[size];
8  #endif
9      elementData[size++] = o;
10 #ifdef TXN
11     l.unlock();
12 #endif
13     fireStackChanged();
14   }
15 #ifdef TXN
16   Lock lock(Object o) {
17     return LockMgr.lockObject(o);
18   }
19 #endif
20   ...
21 }
```

(a) Originalquelltext

```
1  class Stack [] {
2    void push([]) {
3      Lock l = lock(o);
4      []
5      l.unlock();
6      []
7    }
8    Lock lock(Object o) {
9      return LockMgr.lockObject(o);
10   }
11   ...
12 }
```

(b) Sicht auf das Feature TXN (ausgeblendeter Code ist markiert mit '[]'; Kontextinformation ist schrägestellt und grau dargestellt)

Abbildung 5: Sichten emulieren Trennung von Belangen.

finden sich die Kontextinformationen etwa in Schnittstellen von Komponenten und Plugins oder Pointcuts von Aspekten.

Mit Sichten können dementsprechend einige Vorteile der physischen Trennung von Belangen emuliert werden. Damit können auch schwierige Probleme bei der Modularisierung wie das „Expression Problem“ [TOHS99] auf natürliche Weise gelöst werden: entsprechender Quelltext erscheint in mehren Sichten.

Das Konzept von Sichten für präprozessorbasierte Implementierungen kann auch leicht erweitert werden, so dass nicht nur Sichten auf einzelne Features, sondern auch editierbare Sichten auf den gesamten Quelltext einer Produktvariante möglich sind. Eine Sicht kann also genau jenen Quelltext anzeigen, der in einer spezifischen Produktvariante für eine Featureauswahl kompiliert würde, und alle Quelltextfragmente von nicht ausgewählten Features ausblenden. Eine solche Sicht ist sinnvoll, um Fehler in einer bestimmten Variante zu suchen oder um das Verhalten mehrer Feature in Kombination (Stichwort Featureinteraktionen [C$^+$03]) zu studieren. Diese Möglichkeiten von Sichten gehen über das hinaus, was bei modularen Implementierung möglich ist; dort müssen Entwickler das Verhalten von Produktvarianten oder Featurekombinationen im Kopf aus mehreren Komponenten, Plugins oder Aspekten rekonstruieren. Besonders wenn mehrere Features auf feingranularer Ebene interagieren (etwa innerhalb einer Methode), können Sichten eine wesentliche Hilfe sein.

Obwohl Sichten viele Nachteile der fehlenden physischen Trennung von Belangen abmildern können, können zugegebenermaßen nicht alle Nachteile beseitigt werden. Wenn Anforderungen wie separate Kompilierung oder modulare Typprüfung von Features bestehen, helfen auch Sichten nicht weiter. In der Praxis können Sichten aber bereits eine große Hilfe darstellen.

Fehleranfälligkeit. Auch Fehler, die bei Präprozessornutzung entstehen können, können mit Werkzeugunterstützung verhindert werden. Wir stellen insbesondere zwei Gruppen von Ansätze vor: disziplinierte Annotationen gegen Syntaxfehler wie in Abbildung 1 und produktlinienorientierte Typsysteme gegen Typfehler wie in Abbildung 2. Auf Fehler im Laufzeitverhalten (z. B. Deadlocks) gehen wir nicht weiter ein, da diese unserer Meinung nach kein spezifisches Problem von Präprozessoren darstellen, sondern bei modularisierten Implementierungen genauso auftreten können.

Disziplinierte Annotationen. Unter disziplinierten Annotationen versteht man Ansätze, welche die Ausdrucksfähigkeit von Annotationen einschränken, um Syntaxfehler zu vermeiden, ohne aber die Anwendbarkeit in der Praxis zu behindern [KAT+09]. Syntaxfehler entstehen im wesentlichen dadurch, dass Präprozessoren Quelltext als reine Zeichenfolgen sehen und erlauben, dass jedes beliebige Zeichen, einschließlich einzelner Klammern, annotiert werden kann. Disziplinierte Annotationen dagegen berücksichtigen die zugrundeliegende Struktur des Quelltextes und erlauben nur, dass ganze Programmelemente wie Klassen, Methoden oder Statements annotiert (und entfernt) werden können. Die Annotationen in Abbildungen 2 und 5a sind diszipliniert, da nur ganze Statements und Methoden annotiert werden. Syntaxfehler wie in Abbildung 1 sind nicht mehr möglich, wenn disziplinierte Annotationen durchgesetzt werden.

Durch die eingeschränkten Ausdrucksmöglichkeiten mag es für bestimmte Aufgaben schwieriger sein, entsprechende Implementierungen zu finden. In einigen Fällen muss dazu der Quelltext umgeschrieben werden, um das gleiche Verhalten auch mit disziplinierten Annotationen zu ermöglichen. Allerdings hat sich herausgestellt, dass in der Praxis disziplinierte Annotationen die Regel darstellen und andere Implementierungen als 'hack' betrachtet werden [BM01, Vit03]. Der Übergang von undisziplinierten zu disziplinierten Annotationen ist typischerweise einfach und die nötigen Änderungen folgen offensichtlichen, einfachen Mustern. Trotz Einschränkung der Ausdrucksfähigkeit sind disziplinierte Annotationen aber immer noch deutlich ausdrucksfähiger als das, was Modularisierungsansätze wie Komponenten, Plugins oder Aspekte bieten [KAK08].

Auf technischer Seite erfordern disziplinierte Annotationen aufwendigere Werkzeuge als undisziplinierte, da der Präprozessor die zugrundeliegende Quelltextstruktur analysieren muss. Werkzeuge für disziplinierte Annotationen können entweder für bestehenden Quelltext prüfen, ob dieser in disziplinierter Form vorliegt, oder sie können (wie in CIDE [KAK08]) bereits in der Entwicklungsumgebung alle Annotationen verwalten und überhaupt nur disziplinierte Annotationen erlauben. Wie in [KAT+09] gezeigt, ist auch die Erweiterung auf weitere Programmiersprachen und deren zugrundeliegende Struktur einfach und kann weitgehend automatisiert werden, wenn eine Grammatik für die Zielsprache vorliegt.

Produktlinienorientierte Typsysteme. Mit angepassten Typsystemen für Produktlinien ist es möglich, alle Produktvarianten einer Produktlinie auf Typsicherheit zu prüfen, ohne jede Variante einzeln zu kompilieren [CP06, KA08]. Damit können viele wichtige Probleme erkannt werden, wie beispielsweise Methoden oder Klassen, deren Deklaration in einigen Produktvarianten entfernt, die aber trotzdem noch referenziert werden (siehe Abbildung 2).

Während ein normales Typsystem prüft, ob es zu einem Methodenaufruf eine passende Methodendeklaration gibt, wird dies von produktlinienorientierten Typsystemen erweitert,

so dass zudem auch geprüft wird, dass in jeder möglichen Produktvariante entweder die passende Methodendeklaration existiert oder auch der Methodenaufruf gelöscht wurde. Wenn Aufruf und Deklaration mit dem gleichen Feature annotiert sind, funktioniert der Aufruf in jeder Variante; in allen anderen Fällen muss die Beziehung zwischen den jeweiligen Annotationen geprüft werden. Erlauben die Annotationen eine Produktvariante, in der der Aufruf, aber nicht die Deklaration vorhanden ist, wird ein Fehler gemeldet.[3] Durch diese erweiterten Prüfungen zwischen Aufruf und Deklaration (und vielen ähnlichen Paaren) wird mit einem Durchlauf die gesamte Produktlinie geprüft; es ist nicht nötig, jede Produktvariante einzeln zu prüfen.

Wie bei Sichten emulieren produktlinienorientierte Typsysteme wieder einige Vorteile von modularisierten Implementierungen. Statt Modulen und ihren Abhängigkeiten gibt es verteilte, markierte Codefragmente und Abhängigkeiten zwischen Features. Das Typsystem prüft dann, dass auch im verteilten Quelltext diese Beziehungen zwischen Features beachtet werden.

Durch die Kombination von disziplinierten Annotationen und produktlinienorientierten Typsystemen kann die Fehleranfälligkeit von Präprozessoren reduziert werden, mindestens auf das Niveau von modularisierten Implementierungsansätzen. Insbesondere produktlinienorientierte Typsysteme haben sich dabei als hilfreich erwiesen und wurden auch auf modulbasierte Ansätze übertragen (z. B. [TBKC07]).

Schwer verständlicher Quelltext. Durch viele textuelle Annotationen im Quelltext kann dieser schwer lesbar werden, wie in Abbildungen 3 und 4 gezeigt. Hauptverantwortlich dafür ist, dass Präprozessoren wie *cpp* zwei Extrazeilen für jede Annotation benötigen (*#ifdef* und *#endif* je in einer neuen Zeile) und nicht Bestandteil der Host-Sprache sind.

Es gibt verschiedene Ansätze, wie die Darstellung und damit die Lesbarkeit verbessert werden kann. Ein erster Ansatz ist, textuelle Annotationen in einer Sprache mit kürzerer Syntax zu verwenden, die auch Annotationen innerhalb einer Zeile erlauben. Eine zweite Verbesserung ist die Verwendung von Sichten, wie oben diskutiert, welche jene Annotationen, die für die aktuelle Aufgabe unwichtig sind, ausblenden kann. Eine dritte Möglichkeit ist es, Annotationen gezielt grafisch abzusetzen, so dass man sie leichter identifizieren kann. Beispiele dafür sind einige Entwicklungsumgebungen für PHP, die verschiedene Hintergrundfarben für PHP- und HTML-Quelltext innerhalb der gleichen Datei verwenden. Schlussendlich ist es sogar möglich, auf textuelle Annotationen komplett zu verzichten und stattdessen Annotationen komplett auf die Repräsentationsschicht zu verlegen, wie in unserem Werkzeug CIDE.

In CIDE gibt es keine textuellen Annotationen, stattdessen werden Hintergrundfarben im Editor zur Repräsentation von Annotationen benutzt [KAK08]. Beispielsweise wird der gesamte Quelltext der Transaktionsverwaltung in Abbildung 6 mit roter Hintergrundfarbe dargestellt. Auf diese Weise kann man den Quelltext (ursprünglich aus Abbildung 3) deutlich kürzer und lesbarer darstellen. Hintergrundfarben wurden dabei inspiriert von

[3] Es gibt viele Möglichkeiten, Beziehungen zwischen Features und Produktvarianten zu beschreiben. Feature-Modelle und aussagenlogische Ausdrücke sind dabei übliche Mechanismen, über die man zudem auch automatisiert Schlüsse ziehen kann [Bat05].

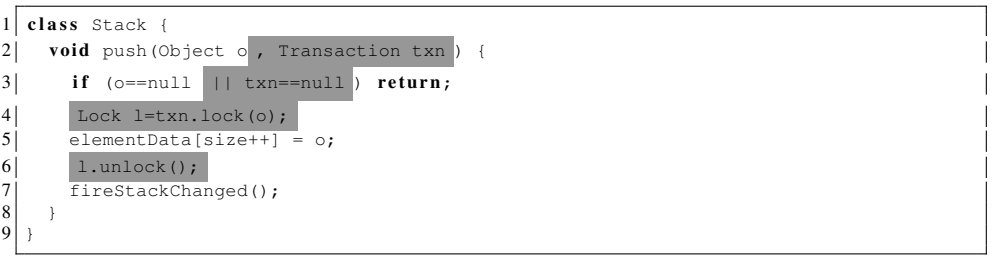


Abbildung 6: Hintergrundfarbe statt textueller Anweisung zum Annotieren von Quelltext.

Quelltextausdrucken auf Papier, die wir zur Analyse mit farbigen Textmarkern (eine Farbe pro Feature) markiert haben. Hintergrundfarben lassen sich leicht in Entwicklungsumgebungen integrieren und sind dort in der Regel noch nicht belegt. Statt Hintergrundfarben gibt es natürlich auch noch viele andere mögliche Darstellungsformen, z. B. farbige Linien neben dem Editor, wie sie in *Spotlight* verwendet [CPR07]. Hintergrundfarben und Linien sind besonders hilfreich bei langen und geschachtelten Annotationen, die bei textuellen Annotationen häufig schwierig nachzuvollziehen sind, besonders wenn das *#endif* einige hundert Zeilen nach dem *#ifdef* folgt wie in Abbildung 1. Uns sind die Begrenzungen von Farben bewusst (z. B. können Menschen nur relativ wenige Farben sicher unterscheiden), aber es gibt ein weites Feld an Darstellungsformen, das noch viel Platz für Verbesserungen lässt.

Trotz aller grafischen Verbesserungen und Werkzeugunterstützung sollte man aber nicht aus dem Auge verlieren, dass der Präprozessor (auch wenn es vielleicht einladend wirkt) nicht als Rechtfertigung dafür dienen darf, Quelltext gar nicht mehr zu modularisieren (mittels Klassen, Paketen, etc.). Sie erlauben den Entwicklern nur mehr Freiheit und zwingen sie nicht mehr, alles um jeden Preis zu modularisieren. Typischerweise wird ein Feature weiterhin in einem Modul oder eine Klasse implementiert, lediglich die Aufrufe verbleiben verteilt und annotiert im Quelltext. Wenn dies der Fall ist, befinden sich auf einer Bildschirmseite Quelltext (nach unseren Erfahrungen mit CIDE) selten Annotationen zu mehr als zwei oder drei Features, so dass man auch mit einfachen grafischen Mitteln viel erreichen kann.

Vorteile von Präprozessoren. Neben allen Problemen haben Präprozessoren auch einige Vorteile, die wir hier nicht unter den Tisch fallen lassen wollen. Der erste und wichtigste ist, dass Präprozessoren ein *sehr einfaches Programmiermodell* haben: Quelltext wird annotiert und entfernt. Präprozessoren sind daher sehr leicht zu erlernen und zu verstehen. Im Gegensatz zu vielen anderen Ansätzen wird keine neue Spracherweiterung, keine besondere Architektur und kein neuer Entwicklungsprozess benötigt. In vielen Sprachen ist der Präprozessor bereits enthalten, in allen anderen kann er leicht hinzugefügt werden. Diese Einfachheit ist der Hauptvorteil des Präprozessors und wahrscheinlich der Hauptgrund dafür, dass er so häufig in der Praxis verwendet wird.

Zweitens sind Präprozessoren *sprachunabhängig* und können für alle Sprachen *gleichförmig* eingesetzt werden. Beispielsweise kann *cpp* nicht nur für C-Quelltext, sondern auch genauso

für Java und HTML verwendet werden. Anstelle eines Tools oder einer Spracherweiterung pro Sprache (etwa AspectJ für Java, AspectC für C, Aspect-UML für UML usw.) funktioniert der Präprozessor für alle Sprachen gleich. Selbst mit disziplinierten Annotationen können Werkzeuge sprachübergreifend verwendet werden [KAT+09].

Drittens, wie bereits angedeutet, verhindern Präprozessoren nicht die traditionellen Möglichkeiten zur Trennung von Belangen. Eine primäre (dominante) Dekomposition etwa mittels Klassen oder Modulen ist weiterhin möglich und sinnvoll. Präprozessoren fügen aber weitere Ausdrucksfähigkeit hinzu, wo traditionelle Modularisierungsansätze an ihre Grenzen stoßen mit querschneidenden Belangen oder mehrdimensionaler Trennung von Belangen [K+97, TOHS99]. Eben solche Probleme können mit verteilten Quelltext und später Sichten auf den Quelltext leicht gelöst werden.

Werkzeuge. Die in diesem Beitrag vorgestellten Verbesserungen für Präprozessoren kommen aus verschiedenen Forschungsarbeiten. Wir haben eigene und verwandte Arbeiten in dem Bereich vorgestellt und zu einer einheitlichen Lösung zusammengeführt. Alle Verbesserungen – Sichten auf Features und Produktvarianten, erzwungene disziplinierte Annotationen, ein Java-Typsystem für Produktlinien und eine visuelle Darstellung von Annotationen – sind in unserem Produktlinienwerkzeug-Prototyp CIDE unter anderem für Java implementiert. In CIDE werden Annotationen statt als textuelle *#ifdef*-Anweisungen direkt in der Entwicklungsumgebung als Mapping zwischen Features und der zugrundeliegenden Quelltextstruktur gespeichert. Es erzwingt daher von vornherein disziplinierte Annotationen und eignet sich (da alle Annotationen leicht zugreifbar verfügbar sind) gut als Testbett für Typsysteme, Sichten und verschiedene Visualisierungen. CIDE steht unter `http://fosd.de/cide` zum Ausprobieren zusammen mit diversen Fallbeispielen zur Verfügung.

4 Zusammenfassung

Unsere Kernmotivation für diesen Beitrag war es zu zeigen, dass Präprozessoren für die Produktlinienentwicklung keine hoffnungslosen Fälle sind. Mit etwas Werkzeugunterstützung können viele der Probleme, für die Präprozessoren kritisiert werden, leicht behoben oder zumindest abgeschwächt werden. Sichten auf den Quelltext können Modularität oder eine Trennung von Belangen emulieren, disziplinierte Annotationen und Typsysteme für Produktlinien können Implementierungsprobleme frühzeitig erkennen und Quelltexteditoren können den Unterschied zwischen Quelltext und Annotationen hervorheben oder Annotationen gar komplett auf die Repräsentationsschicht verlagern. Obwohl wir nicht alle Probleme der Präprozessoren lösen können (beispielsweise ist ein separates Kompilieren der Features weiterhin nicht möglich), haben Präprozessoren auch einige Vorteile, insbesondere das einfache Programmiermodell und die Sprachunabhängigkeit. Zusammen nennen wir diese Verbesserungen „Virtuelle Trennung von Belangen“, da sie, obwohl Features nicht tatsächlich physisch in Module getrennt werden, dennoch diese Trennung durch Werkzeugunterstützung emulieren.

Als Abschluss möchten wir noch einmal betonen, dass wir selber nicht endgültig entschei-

den können, ob eine echte Modularisierung oder eine virtuelle Trennung langfristig der bessere Ansatz ist. In unserer Forschung betrachten wir beide Richtungen und auch deren Integration. Dennoch möchten wir mit diesem Beitrag Forscher ermuntern, die Vorurteile gegenüber Präprozessoren (üblicherweise aus Erfahrung mit *cpp*) abzulegen und einen neuen Blick zu wagen. Entwickler in der Praxis, die zurzeit Präprozessoren verwenden, möchten wir im Gegenzug ermuntern, nach Verbesserungen Ausschau zu halten bzw. diese von den Werkzeugherstellern einzufordern.

Danksagung. Wir danken Jörg Liebig, Marko Rosenmüller, Don Batory und Jan Hoffmann für ihre Unterstützung und die Quelltextbeispiele aus Berkeley DB und Femto OS. Sven Apel wurde durch die DFG unterstützt, Projektnummer AP 206/2-1.

Literatur

[AK09] Sven Apel und Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, Jgg. 3714 of *LNCS*, Seiten 7–20. Springer, 2005.

[BCK98] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl und Klaus Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. Dpunkt Verlag, 2004.

[BM01] Ira Baxter und Michael Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, Seiten 281–290. IEEE, 2001.

[C+03] Muffy Calder et al. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.

[CP06] Krzysztof Czarnecki und Krzysztof Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, Seiten 211–220. ACM, 2006.

[CPR07] David Coppit, Robert Painter und Meghan Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. Software Eng. (ICSE)*, Seiten 754–757. IEEE, 2007.

[EBN02] Michael Ernst, Greg Badros und David Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.

[Fav97] Jean-Marie Favre. Understanding-In-The-Large. In *Proc. Int'l Workshop on Program Comprehension*, Seite 29. IEEE, 1997.

[HKW08] Florian Heidenreich, Jan Kopcsek und Christian Wende. FeatureMapper: Mapping Features to Models. In *Comp. Int'l Conf. Software Engineering (ICSE)*, Seiten 943–944. ACM, 2008.

[JDV04] Doug Janzen und Kris De Volder. Programming with Crosscutting Effective Views. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, Jgg. 3086 of *LNCS*, Seiten 195–218. Springer, 2004.

[K+90] Kyo Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Bericht CMU/SEI-90-TR-21, SEI, 1990.

[K+97] Gregor Kiczales et al. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, Jgg. 1241 of *LNCS*, Seiten 220–242. Springer, 1997.

[KA08] Christian Kästner und Sven Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, Seiten 258–267. IEEE, 2008.

[KAK08] Christian Kästner, Sven Apel und Martin Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Eng. (ICSE)*, Seiten 311–320. ACM, 2008.

[KAT+09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann und Don Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, Jgg. 33 of *LNBIP*, Seiten 175–194. Springer, 2009.

[KS94] Maren Krone und Gregor Snelting. On the Inference of Configuration Structures from Source Code. In *Proc. Int'l Conf. Software Eng. (ICSE)*, Seiten 49–57. IEEE, 1994.

[KTA08] Christian Kästner, Salvador Trujillo und Sven Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*. Lero, 2008.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, Jgg. 1241 of *LNCS*, Seiten 419–443. Springer, 1997.

[RALS09] Marko Rosenmüller, Sven Apel, Thomas Leich und Gunter Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 68(12):1493–1512, 2009.

[SC92] Henry Spencer und Geoff Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, Seiten 185–198, 1992.

[Sel08] Margo Seltzer. Beyond Relational Databases. *Commun. ACM*, 51(7):52–58, 2008.

[SGC07] Nieraj Singh, Celina Gibbs und Yvonne Coady. C-CLR: A Tool for Navigating Highly Configurable System Software. In *Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Seite 9. ACM, 2007.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin und William Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, Seiten 95–104. ACM, 2007.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison und Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. Software Eng. (ICSE)*, Seiten 107–119. IEEE, 1999.

[TSSPL09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat und Daniel Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. GPCE Workshop on Feature-Oriented Software Dev. (FOSD)*, Seiten 81–86, New York, NY, 2009. ACM.

[Vit03] Marian Vittek. Refactoring Browser with Preprocessor. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, Seiten 101–110. IEEE, 2003.

Featuremodellbasiertes und kombinatorisches Testen von Software-Produktlinien

Sebastian Oster, Philipp Ritter, Andy Schürr

{oster, pritter, schuerr}@es.tu-darmstadt.de

Abstract: Software-Produktlinien-Entwicklung bietet eine systematische Wiederverwendung von Software-Artefakten. Auf Grund der Tatsache, dass viele Produkte aus einer Produktlinie abgeleitet werden können, ist es unerlässlich Testverfahren zu entwickeln, die zum einen eine möglichst vollständige Abdeckung von allen möglichen Produkten sicherstellen und zum anderen weniger aufwändig sind, als jedes Produkt einzeln zu testen. In diesem Beitrag wird die Entwicklung eines neuen Algorithmus zur Auswahl einer Menge von Produktinstanzen als Testkandidaten beschrieben. Dieser vereint Techniken der Modelltransformation, des kombinatorischen Testens und des Lösens binärer Constraintsysteme vermittels Forward Checking.

1 Einleitung

Software-Produktlinien-Entwicklung ist eine Methode zur systematischen Wiederverwendung von Software-Artefakten. Dabei wird eine Gruppe von Software-Applikationen oder -Produkten beschrieben, die sich gewisse gemeinsame Eigenschaften teilen und somit spezielle Anforderungen einer Domäne erfüllen [PBvdL05].

Software-Produktlinien (SPL) werden in verschiedenen Domänen, unter anderem im Automobil-Bereich, erfolgreich eingesetzt, um die Kosten der Entwicklung und Wartung zu senken und die Qualität der einzelnen Produkte zu steigern. Im Automobil-Bereich werden die Entwickler derzeit mit einer Situation konfrontiert, in der (1) 50-70 ECUs (Electronic Control Units) für die Steuerung diverser Funktionen in einem Auto verbaut werden und (2) jede dieser ECUs bis zu 10.000 verschiedene Konfigurationsmöglichkeiten besitzt. Daraus ergeben sich Millionen von verschiedenen Variationen. Bei gewissen Modellen einiger Hersteller rollen bereits Autos vom Band, die alle unterschiedliche Softwarekonfigurationen enthalten. In der Praxis werden Software-Produktlinien häufig dadurch validiert, dass jede generierte Produktinstanz wie ein einzelnes Softwareprodukt getestet wird [TTK04]. Diese Methode bietet den Vorteil, dass man einen Großteil der Variabilität vernachlässigen und bekannte und bewährte Methoden aus dem Software-Engineering einsetzen kann. Dies ist jedoch in vielen Fällen, wie das Beispiel aus dem Automobil-Bereich zeigt, auf Grund der hohen Anzahl an möglichen Produkten nicht mehr ausführbar. Aus diesem Grund wird nach Lösungen gesucht, die den Testaufwand reduzieren.

Erstrebenswert ist die Entwicklung einer Testmethode, die den Testaufwand minimiert, dabei jedoch nicht auf den Vorteil, den das individuelle Produkttesten mit sich bringt,

verzichtet. Ziel wäre die Identifizierung einer minimalen repräsentativen Menge von Produkten. Da dies ein NP-vollständiges Problem darstellt [Sch07] kann eine solche Menge nur durch Heuristiken approximiert werden. Das erfolgreiche Testen dieser Menge soll mit hoher Wahrscheinlichkeit garantieren, dass der Test aller weiteren Produkte ebenfalls erfolgreich ist.

In dieser Arbeit wird untersucht, inwieweit sich kombinatorisches Testen als Grundlage einer solchen Heuristik eignet und wie kombinatorisches Testen auf Featuremodelle angewendet werden kann. Kombinatorisches Testen wird zur Reduzierung des Testaufwands bei der Auswahl von Testparametern bereits erfolgreich eingesetzt [CDKP94, LT98]. Da prinzipiell Features der Parametrisierung von SPLs dienen, scheint es vielversprechend, kombinatorisches Testen auf SPLs anzuwenden [McG01, CDS07]. Dabei werden Features als Parameter einer SPL interpretiert. Fokus dieses Beitrags ist die Bewältigung der technischen Herausforderungen um kombinatorisches Testen auf Featuremodelle anzuwenden.

Der hier vorgestellte Ansatz wurde im Rahmen des BMBF-Projektes feasiPLe entwickelt [fC06] und ist eine Weiterführung der Arbeit in [OS09]. Das Verfahren wurde auf zwei verschiedene Arten realisiert. Zum einen wurde ein eigener Feature-Modell-Editor implementiert, der die Vorgehensweise auf Basis von Modelltransformation mit Fujaba/MOFLON [AKRS06] realisiert. Zum anderen wurde im Rahmen des feasiPLe-Projekts ein Plugin für das Tool pure::variants der Firma pure-systems erstellt, welches die Methodik umsetzt. In diesem Beitrag wird ausschließlich das pure::variants Plugin beschrieben.

Für die Beschreibung der Vorgehensweise ist die Arbeit wie folgt aufgebaut. In Abschnitt 2 werden themenverwandte Arbeiten aufgeführt und erläutert. Anschließend werden in Abschnitt 3 Grundlagen zu Software-Produktlinien und kombinatorischem Testen beschrieben. Abschnitt 4 widmet sich dem Konzept dieses Beitrags. In diesem werden zum einen die notwendigen Schritte beschrieben, um die Anwendung von kombinatorischem Testen auf Featuremodellen zu ermöglichen und zum anderen der eigens für diese Anwendung entwickelte Algorithmus zur Realisierung des kombinatorischen Testens vorgestellt. Jeder einzelne Schritt wird dabei anhand eines Anwendungsbeispiels veranschaulicht. Eine Beschreibung der Implementierung mit pure::variants ist in Abschnitt 4.3 aufgeführt. Abschließend wird dieser Beitrag in Sektion 5 zusammengefasst und ein Ausblick über weitere Forschungsaktivitäten dargestellt.

2 Verwandte Arbeiten

Testen von Software-Produktlinien

Für das Testen von Software-Produktlinien existieren viele verschiedene Ansätze, mit der Zielsetzung Fehler zu identifizieren, um eine bestimmte Qualität der Software zu gewährleisten. Eine Zusammenfassung ist in [TTK04] aufgeführt. Die Autoren unterteilen die Testansätze in vier Kategorien nach denen Testen in den Entwicklungsprozess eingebunden werden kann.

Product-by-product testing: Als product-by-product testing wird die Testmethodik bezeichnet, die jede einzelne Produktinstanz individuell testet und dabei auf Testverfahren zurückgreift, die in der Software-Engineering-Community bekannt sind. Wie bereits in der Einleitung erwähnt, ist es in vielen Bereichen nicht mehr möglich alle Produkte einzeln zu

testen. Eine Möglichkeit, die Menge an Produkten zu reduzieren, ist die Generierung einer repräsentativen Menge von Produkten für die SPL. Nachteil dieses Ansatzes ist, dass die Bestimmung einer minimalen Testmenge von Produkten ein NP-vollständiges Problem darstellt [Sch07]. In der Regel wird also nicht die minimale Menge von Produkten bestimmt, sondern es werden zu viele, sich äquivalent verhaltende Produktinstanzen erzeugt.
Inkrementelles Testen: In diesem Testverfahren werden, ähnlich wie im product-by-product testing, die Produkte einzeln getestet, jedoch wird der Testaufwand unter Verwendung von Regressionstestverfahren verringert. Dabei wird ein spezielles Produkt basierend auf speziellen Kriterien ausgewählt und ausführlich getestet. Daran anschließend wird ein weiteres Produkt getestet, welches eine sehr hohe Ähnlichkeit im Bezug zu der Funktionalität zum ersten Produkt hat. Die Grundidee ist, dass nur die veränderten Komponenten ausführlich getestet werden müssen, da die Gemeinsamkeiten bereits mit dem ersten Produkt getestet wurden. Eine Schwierigkeit bei dieser Testmethode ist die Tatsache, dass entschieden werden muss, welches Produkt initial ausführlich getestet wird und welche Teile dann nicht mehr getestet werden müssen [Con04, McG01].
Wiederverwendung von Test Assets: Bei dieser Testmethodik steht die Aufteilung in Domain- und Application-Engineering im Vordergrund. Ziel ist es, im Domain-Engineering wiederverwendbare Testartefakte zu entwickeln, die dann für das Testen der Produkte verwendet werden können. Diese Testartefakte werden im Application-Engineering-Prozess an das jeweilige Produkt angepasst. Diesem Ansatz haben sich viele SPL-Testverfahren verschrieben. Bertolino und Gnesi [BG03] erweitern die Category-Patitioning-Methode um Variabilität um diese wiederverwenden zu können. McGregor [McG01] erstellt abstrakte Domänentestfälle, die spezialisiert auf einzelne Applikationen angewendet werden. Ähnlich dazu werden in den Testverfahren CADeT [Oli08] und ScenTED [RKPR05] wiederverwendbare Testmodelle im Domain-Engineering generiert und für jede Variante adaptiert. Nebut et al. generieren Tests auf Basis von Use Cases [NFLTJ04]. Um diese wiederverwenden zu können, wird Variabilität integriert. Weißleder et al. nutzen eine Statemachine um die Funktionalität der gesamten Produktlinie zu modellieren [WSS08]. Diese wird dazu verwendet um produktspezifische Testfälle zu generieren.
Aufteilung nach Verantwortlichkeit: Bei diesem Testverfahren werden den Domain- und Application-Engineering Prozessen die verschiedenen Test-Level (Unit, Integration, System und Acceptance) zugewiesen. Dabei könnten z.B. Unit Tests im Domain-Engineering und die restlichen Test-Level im Application-Engineering ausgeführt werden. Ein weiterer Vorschlag erfolgt in [JhQJ08], in dem für Domain- und Application-Engineering jeweils ein V-Modell mit allen Test-Leveln durchlaufen wird.

Allen Ansätzen ist gemeinsam, dass Algorithmen zur Bestimmung einer Sequenz oder Menge zu testender Produktinstanzen benötigt wird, um nicht alle Produkte testen zu müssen. Eine Arbeit, die sich mit diesem Thema befasst ist [Sch07]. Die Autorin generiert für jede Anforderung eine repräsentative Menge. Ein weiterer Algorithmus wird in dieser Arbeit vorgestellt, welcher auf dem Prinzip des kombinatorischen Testens basiert. Die in diesem Ansatz generierten Produktinstanzen können durch die aufgezählten Methodiken getestet werden.

Kombinatorisches Testen

Um die Korrektheit eines Programms gewährleisten zu können, muss jede mögliche Kombination aller verfügbaren Eingabewerte getestet werden [Bei90]. Dies ist auf Grund der Komplexität der meisten Programme und der daraus resultierenden Kombinationsmöglichkeiten jedoch fast nie realisierbar. Selbst bei einer Software mit lediglich 10 Eingabefeldern mit jeweils 3 Möglichkeiten wären schon $3^{10} = 59.049$ Testfälle abzuarbeiten. Wird die Anzahl der Möglichkeiten auf 5 erhöht, ergeben sich bereits $5^{10} = 9.765.625$ Testfälle. Zahlreiche Konzepte existieren, welche die Testfallanzahl anhand von kombinatorischen Testen minimieren während die Testabdeckung möglichst groß bleibt. Für die Anwendung auf Software-Produktlinien ist es unerlässlich, dass Abhängigkeiten zwischen Parametern, aber auch zwischen Parameterwerten, beachtet werden können. Eine Zusammenfassung einiger solcher Algorithmen ist in [CDS07] gegeben. Dabei wird zwischen mathematischen-, Greedy- und meta-heuristischen Verfahren differenziert. Diese Übersicht zeigt auch, welche dieser Algorithmen in der Lage sind zusätzliche Abhängigkeiten zwischen den Parametern zu verarbeiten. Eine Anwendung des kombinatorischen Testens ist das paarweise Testen. Diese Methode nutzt die Erkenntnis, dass der Großteil aller Softwarefehler aus einzelnen Datenwerten oder der Interaktion zweier Datenwerte resultiert [SM98]. Daher werden lediglich die Eingabefelder paarweise mit jedem anderen Eingabefeld getestet, wobei zwischen diesen zwei Feldern weiterhin alle möglichen Kombinationen gebildet werden. Sehr bekannte Arbeiten sind dabei AETG [CDKP94] oder IPO [LT98]. AETG und IPO erreichen ihr Ziel, alle paarweisen Kombinationen in möglichst wenig Testfällen unterzubringen, auf unterschiedliche Weise, wobei der AETG oftmals eine etwas geringere Testfallanzahl als der IPO erzeugt. Da es sich bei AETG jedoch um ein kommerzielles Projekt handelt, dessen Realisierungsdetails nicht bekannt sind, dient in dieser Arbeit der frei verfügbare IPO als Grundlage für den in Abschnitt 4.2 vorgestellten Algorithmus. Eine kritische Diskussion des paarweisen Testens ist in [BS04] aufgeführt.

3 Grundlagen

Als Anwendungsbeispiel dient ein Ausschnitt einer Handy-Produktlinie, die für die Lehre und Forschung entwickelt wird. Alle Funktionalitäten werden auf Basis der Google Android-Plattform realisiert. Für die Veranschaulichung wird eine Produktlinie gewählt, die über Grundfunktionen, Datenübertragungsmöglichkeiten sowie einige Extras verfügt. Zu den Grundfunktionen zählen das Telefonieren und das Versenden von Nachrichten. Zum Datenaustausch stehen der Produktlinie WLAN, Bluetooth und UMTS zur Verfügung. Mögliche Extras sind ein integrierter MP3-Player und eine Kamera. Diese Produktlinie ermöglicht die Ableitung von 26 gültigen Produktvarianten.

Software-Produktlinien und Featuremodelle

Featuremodelle werden für die explizite Darstellung der Variabilität und der Gemeinsamkeiten innerhalb der SPL-Entwicklung verwendet. Sie bestehen aus hierarchisch angeordneten Features, welche Systemeigenschaften beschreiben, die relevant für einige Stakeholder sind [CHE05]. In [Bos00] werden Features als „eine logische Gruppe von Anforderungen“ definiert. Im Domain-Engineering wird dazu das gesamte Featuremodell

aufgestellt. Im Application-Engineering wird nur noch mit einem Teilbaum des Featuremodells gearbeitet, der ein einzelnes Produkt der SPL beschreibt. Featuremodelle sind intuitiv verständlich und bieten neben der hierarchischen Struktur weitere Notationen und Abhängigkeiten um die Auswahlmöglichkeiten von Features zu kontrollieren. Das erste Featuremodell wurde von Kang et al. [KCH+90] als Teil der FODA-Machbarkeitsstudie eingeführt. In diesem konnten Pflicht-, Optional- und Alternativ-Features definiert werden. Im Laufe der Zeit wurden Erweiterungen für das Featuremodell entwickelt und diskutiert. Einige Erweiterungen werden in [CHE05] erfasst.

In dieser Arbeit wird die Notation aus [KCH+90] verwendet und durch grafische Exclude- und Require-Kanten ergänzt. Exclude-Beziehungen werden als Kante zwischen zwei sich ausschließende Features eingezeichnet. Unidirektionale Require-Kanten verbinden Features, bei denen die Pfeilrichtung angibt, welches Feature das andere für die Erstellung eines Produktes benötigt. Weiterhin wird eine Oder-Gruppe eingeführt. Features innerhalb dieser Oder-Gruppe können in beliebigen Kombinationen in ein Produkt integriert werden. Einzige Restriktion ist, dass mindestens ein Feature dieser Gruppe ausgewählt wird. Abbildung 1 zeigt das Featuremodell der Handy-Produktlinie. Wie bereits erwähnt, bietet

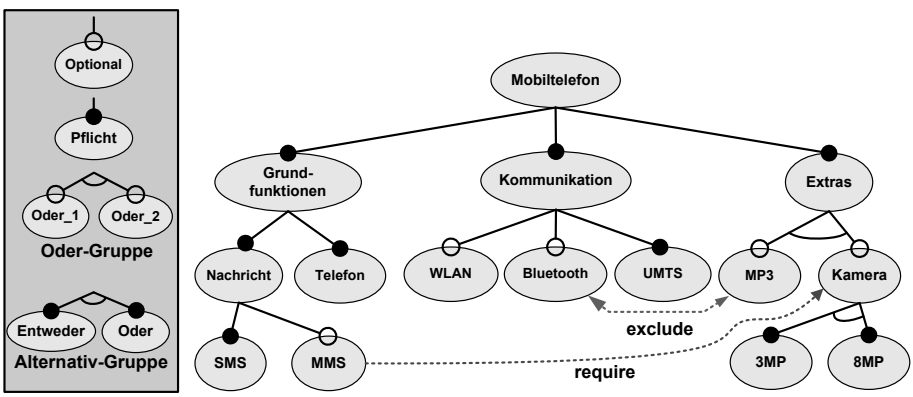


Abbildung 1: Featuremodell des Anwendungsbeispiels

das Handy verschiedene Grundfunktionen, Kommunikations-Funktionalitäten und Extras. Zu den Grundfunktionen zählt das Telefonieren und das Versenden von Nachrichten. Das Telefonieren und das Versenden von SMS ist in jedem Produkt integriert. Das Versenden von MMS ist jedoch optional auswählbar und dementsprechend durch einen unausgefüllten Kreis am Ende der Kante markiert. Für die Kommunikation stehen WLAN und Bluetooth als optionale Features zur Verfügung. UMTS ist in jeder Instanz dieser Produktlinie enthalten. Als Extras stehen ein integrierter MP3-Player und eine integrierte Kamera zur Verfügung. Diese beiden Features sind in einer Oder-Gruppe eingebunden, was zur Folge hat, dass mindestens eines dieser Extras verwendet werden muss. Die Kamera kann entweder einen 3-Megapixel-Sensor oder einen 8-Megapixel-Sensor ansteuern.

Zusätzlich muss bei der Herleitung von Produkten eine Require- und eine Exclude-Abhängigkeit beachtet werden. Wenn MMS-Nachrichten verschickt werden sollen, dann muss eine Kamera integriert werden. MP3-Player und Bluetooth Anbindung schließen sich aus.

Diese Restriktionen sind erdacht, zeigen aber, dass bei Featuremodellen nicht nur „intuitive“ Abhängigkeiten bestehen, sondern durchaus welche, die aus Marketing Sicht Sinn ergeben.

4 Konzept

Features dienen der Parametrisierung von Software-Produktlinien. Die An- und Abwahl von Features entscheidet über die Ausprägung der abgeleiteten Produkte. Daher scheint es vielversprechend zu sein, bekannte Methoden aus dem Software Test auf Software Produktlinien anzuwenden, die sich der Reduzierung des Testaufwands anhand der Parametrisierung widmen [CDS07]. Einer dieser Ansätze ist das kombinatorische Testen, bei dem nur spezielle Kombinationen und nicht alle möglichen Kombinationen von Parametern getestet werden. Es existiert kein Ansatz, der die Anwendung des kombinatorischen bzw. paarweisen Testen auf Featuremodelle beschreibt. Auf Grund der Tatsache, dass Algorithmen des paarweisen Testens Eingabe-Parameter mit dazugehörigen Parameterwerten erwarten, ist es diesen nicht möglich die hierarchische Struktur inklusive Constraints eines Featuremodells zu verarbeiten.

Für die Anwendung des paarweisen Testen auf Featuremodelle sind intuitiv zwei Lösungswege möglich. Die Algorithmen müssen entweder erweitert werden, damit sie die komplexe Struktur und internen Abhängigkeiten eines Featuremodells verarbeiten können oder das Featuremodell muss umstrukturiert werden, sodass die semantische Äquivalenz erhalten bleibt und sich Parameter und Parameterwerte für die Algorithmen ergeben. In dieser Arbeit werden beide Lösungswege kombiniert. Die Struktur des Featuremodells wird durch eine Tiefenreduktion verändert, wobei Parameter und Parameterwerte gebildet werden, auf denen Algorithmen des kombinatorischen Testens arbeiten können. Zudem wird ein Algorithmus entwickelt, ähnlich zum IPO-Algorithmus, der Constraints verarbeiten kann.

4.1 Tiefenreduktion des Featuremodells

Um aus dem Featuremodell Parameter und Parameterwerte zu extrahieren, wurde eine Tiefenreduktion entwickelt, welche aus folgenden zwei Schritten besteht:
1. alle Features mit ihrer zugehörigen Notation werden iterativ nach oben gezogen. Ebenso bleiben die binären Constraints in Form von Require- und Exclude-Kanten erhalten.
2. anschließend werden jedem Feature die korrespondierenden Belegungsmöglichkeiten als Parameterwert angehängt.
Die Tiefenreduktion setzt sich aus mehreren Modelltransformationsregeln zusammen, die das iterative Hochziehen von Features realisieren. Diese Modelltransformationen wurden einerseits mit MOFLON prototypisch realisiert, andererseits als eigenständige Java-Implementierung als Plugin für pure::variants entwickelt. Diese Regeln werden jeweils auf Teilbäume mit ihren assoziierten Constraints angewendet. Diese Teilbäume bestehen immer aus drei Ebenen von Knoten: dem Großvaterknoten, dem Vaterknoten und dem Kind-

knoten. Die Kindknoten werden auf die Ebene des Vaterknotens gehoben, also mit auf die Ebene unter den Großvaterknoten gesetzt. Neben der Hierarchie spielen auch die verschiedenen Notationen der Features eine essentielle Rolle. Optional-Features können nicht genauso hochgezogen werden, wie Pflicht-Knoten usw.. Abbildung 2 zeigt die Hochzieh-

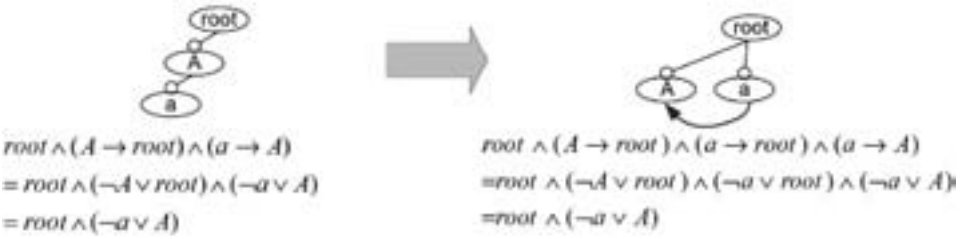


Abbildung 2: Hochziehregel: Optionaler Kindknoten und optionaler Vaterknoten

regel für einen optionalen Kindknoten **a** und einem optionalen Vaterknoten **A**. Die Notation des Großvaterknotens **root** ist zu diesem Zeitpunkt irrelevant. Der Kindknoten kann als optionaler Knoten neben den Vaterknoten gezogen werden. Da jedoch sichergestellt werden muss, dass **a** nur dann ausgewählt werden kann, wenn auch der Vaterknoten **A** ausgewählt ist, wird eine Require-Kante von **a** auf **A** gezogen. Um die semantische Äquivalenz zwischen dem Ursprungs- und dem hochgezogenen Featuremodell zu überprüfen, können beide in einen logischen Ausdruck nach den Regeln in [CW07] übersetzt werden. Dieser Äquivalenznachweis ist ebenfalls in Abbildung 2 aufgeführt.

Für Pflicht-Kindknoten gilt immer die selbe Regel: das Pflicht-Feature wird in den Vaterknoten integriert, da der Vaterknoten nie ohne den Pflicht-Kindknoten existieren kann. Für alle anderen Notationen ergeben sich individuelle Hochziehregeln. Aus jeder Vater-Kind Kombination resultiert eine eigene Regel: 12 Regeln + 1 Regel für Pflicht-Kindkonten → 13 Regeln. Bei der Ausführung der Regeln kommen zusätzliche Require- und Exclude-Kanten hinzu. Diese müssen zusammen mit den ursprünglichen Require- und Exclude-Kanten erhalten bleiben. Diese Regeln werden iterativ ausgeführt, bis der Großvaterknoten dem Wurzelknoten des Featuremodellels entspricht. Angewendet auf das Anwendungsszenario ergibt sich das in Abbildung 3 dargestellte, flache Featuremodell.

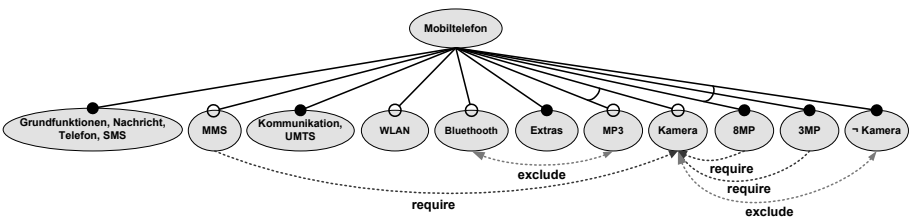


Abbildung 3: Tiefenreduziertes Featuremodell des Anwendungsbeispiels

Ebenso wie bei den einzelnen Regeln kann auch hier zur Sicherheit die semantische Äquivalenz zwischen dem Ursprungs-Featuremodell und dem flachen Featuremodell durch

Übersetzen in einen logischen Ausdruck nachgewiesen werden. Beide Modelle ergeben den selben logischen Ausdruck:

$$
\begin{aligned}
&Mobiltelefon \wedge Grundfunktionen \wedge Nachricht \wedge Telefon \wedge SMS \\
&\wedge Kommunikation \wedge UMTS \wedge Extras \wedge (MP3 \vee Kamera) \wedge \\
&\vee [(\neg 3MP \wedge \neg 8MP \wedge \neg Kamera) \vee (3MP \wedge \neg 8MP \wedge Kamera) \\
&\vee (\neg 3MP \wedge 8MP \wedge Kamera)] \wedge (\neg MMS \vee Kamera) \\
&\wedge [(\neg Bluethooth \wedge MP3) \vee (\neg MP3 \wedge Bluetooth)]
\end{aligned}
\tag{1}
$$

Nach der Tiefenreduktion entspricht das flache Featuremodell einer Menge von Parametern mit ihren zugehörigen Wertebereichen. Jedes Feature wird für das paarweise Testen als ein Parameter behandelt. Zusätzliche Abhängigkeiten müssen dabei beachtet werden. Letzter Schritt für die Vorbereitung ist die Extraktion von Parameterwerten für diese Parameter.

Die Parameterwerte entsprechen den möglichen Belegungen, die ein Parameter einnehmen kann. Ein optionales Feature hat beispielsweise die Werte „vorhanden“ und „nicht vorhanden“. Pflicht-Knoten haben immer nur einen Wert, nämlich vorhanden. Bei Oder- und Alternativ-Gruppen ist diese Belegung etwas komplexer. In einer Oder-Gruppe sind alle Featurekombinationen möglich, nur gibt es die Einschränkung, dass immer mindestens eines der Features ausgewählt wird. Bei Alternativ-Gruppen muss genau ein Gruppenelement ausgewählt werden wobei die anderen Features ausgeschlossen werden.

Um die Parameterwerte in dem flachen Featuremodell abzubilden, wird eine zusätzliche Ebene eingeführt. In Abbildung 4 ist das flache Featuremodell mit einer zusätzlichen Ebene abgebildet, die die Parameterwerte der Parameter enthält. Die Notation der Parameter wird in Pflicht-Notation umgewandelt, da die Ursprungs-Notation nun durch die Wertebelegungen repräsentiert wird. Das optionale Feature MMS ist nun ein Parameter mit den zwei mögliche Werten MMS und ¬ MMS. Da nur genau einer dieser Wertebelegungen bei der Instanziierung einer Produktvariante und beim Test möglich ist, sind diese Werte als Alternativ-Gruppe markiert.

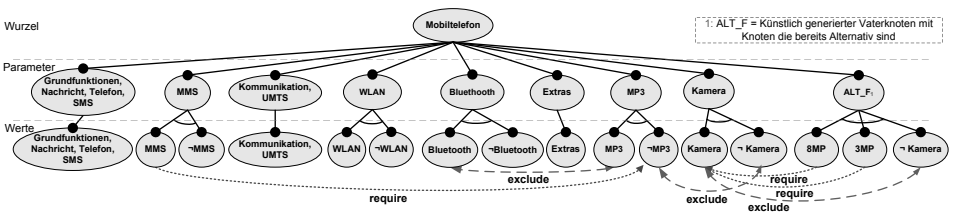


Abbildung 4: Tiefenreduziertes Featuremodell mit Parameterwerten

Für die Erhaltung von Require- und Exclude-Abhängigkeiten müssen diese auf die Parameterwerte übertragen werden. Ein Require-Constraint, welcher zum Beispiel auf das optionale Feature Kamera verweist, muss nun auf den Parameterwert Kamera verschoben werden.

Die Darstellung von Parametern und Parameterwerten skaliert in dieser Darstellungsart nicht sehr gut für große Featuremodelle. Bei der Integration in pure::variants wird dieses Modell nie angezeigt, sondern dient nur für die interne Weiterverarbeitung. Eine Anzeige des flachen Featuremodells ist nicht geplant und auch nicht nötig, da keine weiteren Vorteile aus dieser Darstellung gewonnen werden können. Für die Veranschaulichung von Parametern und Parameterwerten ist die Darstellung durch eine Tabelle vorzuziehen. Eine Überführung des flachen Featuremodells in eine Tabelle ist ohne Weiteres möglich.

4.2 Produktinstanzerzeugung durch paarweises Testen

Nachdem das Featuremodell in Parameter und dazugehörige Werte überführt wurde, können Algorithmen zur Realisierung von paarweisem Testen angewendet werden. Grundidee hinter dem Generierungsalgorithmus ist die Vermutung, dass Fehler in der Integration von Features in 2-er Kombinationen auftreten. Wie der Vergleich in [CDS07] zeigt, können nur wenige Algorithmen zusätzliche Constraints zwischen Parametern und ihren Werten verarbeiten. Diese sind jedoch nicht in ihrer Funktionalität offen gelegt. Daher wurde auf Basis des IPO-Algorithmus [LT98] ein Verfahren entwickelt, welches paarweise Kombinationen unter Beachtung von ggf. existierenden Constraints erzeugt. Der Pseudocode ist in Abbildung 5 aufgeführt. Dazu wird zunächst eine Liste aller möglichen paarweisen

```
01 begin
02   //M is a list of all pairs which have to be covered by a testcase
03   M := {(v_i,v_k) | v_i and v_k not exclude each other,
          v_i and v_k are values of parameters p_i and p_k | k!=i};
04   T := {};           // list of test cases
05   begin
06     while M != {}    //M is not empty
07     begin
08       P := {(v_0,v_1) | (v_0,v_1) ε  M};  //testpair taken from M;
09       t := valid test case that covers at least pair P
               (determined by forward checking);
10       M':= {contains all pairs covered in t};
11       M := M - M';   //remove covered pairs from list
12       T := T + t;    //add new test t to list of tests
13     end
14   end
15 end
```

Abbildung 5: Pseudocode der Produktinstanziierung

Kombinationen erstellt (Zeile 3). Um gültige Produktinstanzen zu bestimmen, wird eine Kombination aus dieser Liste als Startwert einer neuen Produktinstanz verwendet und anschließend mit Hilfe eines Forward-Checking-Algorithmus [HE80] für jeden Parameter ein Wert bestimmt wird (Zeile 9). Eine gültige Produktinstanz liegt vor, wenn für jeden Parameter ein Wert gefunden wurde, der keinen anderen in der Produktinstanz befindlichen Wert durch eine Exclude-Kante ausschließt sowie alle Require-Kanten erfüllt sind. Ist eine solche Produktinstanz generiert worden, werden alle in der Produktinstanz befindlichen

paarweisen Kombinationen (Zeile 10) aus der Liste der benötigten Kombinationen entfernt (Zeile 11) und es werden für die übrig gebliebenen Kombinationen weitere Produktinstanzen erzeugt (Zeile 6). Kann der Forward-Checking-Algorithmus aus dem Startwert keine gültige Produktinstanz erzeugen, wird diese Kombination verworfen.

Das Ergebnis dieses Algorithmus ist eine Menge von Produktinstanzen, die z.B. genutzt werden können, um die SPL durch die Ausführung von Systemtests auf jeder Instanz zu validieren. Die Anwendung des Algorithmus' auf die Handy-SPL resultiert in 8 Produktinstanzen, welche in Abbildung 6 tabellarisch aufgeführt werden.

G,N,T,S	Ko,UMTS	Extras	MMS	WLAN	BT	MP3	Kamera	ALF_F
G,N,T,S	Ko,UMTS	Extras	MMS	WLAN	BT	¬MP3	Kamera	8MP
G,N,T,S	Ko,UMTS	Extras	¬MMS	¬WLAN	¬BT	MP3	¬Kamera	¬Kamera
G,N,T,S	Ko,UMTS	Extras	¬MMS	WLAN	¬BT	¬MP3	Kamera	3MP
G,N,T,S	Ko,UMTS	Extras	MMS	¬WLAN	¬BT	MP3	Kamera	3MP
G,N,T,S	Ko,UMTS	Extras	¬MMS	¬WLAN	BT	¬MP3	Kamera	8MP
G,N,T,S	Ko,UMTS	Extras	¬MMS	WLAN	¬BT	MP3	¬Kamera	¬Kamera
G,N,T,S	Ko,UMTS	Extras	¬MMS	WLAN	BT	¬MP3	Kamera	3MP
G,N,T,S	Ko,UMTS	Extras	¬MMS	WLAN	¬BT	MP3	Kamera	8MP

Abbildung 6: Ergebnis der Produktinstanziierung auf der Handy-SPL

4.3 Integration in pure::variants

pure::variants, ein weit verbreitetes und mächtiges Tool für das Variantenmanagement, speichert Featurebäume in einer XML-Struktur. Somit ist es möglich, durch XML-Parser die von pure::variants erzeugten Featurebäume einzulesen, in eine eigene Datenstruktur zu übertragen und ggf. wieder als ein, für pure::variants lesbares, Featuremodell abzuspeichern. Durch die Unterstützung von pure-systems war es möglich, den in pure::variants integrierten Parser zu nutzen, womit auf die Umsetzung eines eigenen Parsers verzichtet werden konnte. Um die nötigen Transformationen, d.h. die Tiefenreduktion und die Erzeugung der Produktinstanzen, möglichst einfach aufrufen zu können, wurde ein Eclipse-Plugin entwickelt, welches im Kontextmenü der Featurebäume den Aufruf der Transformationen ermöglicht. Die so erzeugten Produktinstanzen werden in einem Unterordner abgelegt und können durch pure::variants betrachtet bzw. weiter verarbeitet werden.

5 Zusammenfassung und Ausblick

Ziel ist, mit Hilfe von kombinatorischem Testen, den Testaufwand für SPLs zu reduzieren. Dabei wird in diesem Beitrag die Grundlage für die Anwendung des kombinatorischen Testens auf ein Featuremodell geschaffen. Es wurde eine Methodik zur Anpassung des Featuremodells vorgestellt, um kombinatorisches Testen anzuwenden. Durch die Tiefenreduktion kann das Featuremodell durch eine einfache Liste innerhalb eines Programms (z.B. pure::variants) dargestellt und verarbeitet werden. Dabei verändert die Tiefenreduktion den Inhalt des Featuremodells nicht sondern ausschließlich die Struktur. Die semantische Äquivalenz kann durch Übersetzen in logische Ausdrücke überprüft werden.

In diesem Beitrag wurde ein Algorithmus für paarweises Testen entwickelt, der sich an

dem IPO-Algorithmus orientiert. Ebenfalls verarbeitet dieser Algorithmus Abhängigkeiten zwischen Parametern und Parameterwerten. Dabei werden zwei Arten von Constraints ausgewertet: bidirektionale Excludes und unidirektionale Requires. Dazu wurde der Algorithmus durch Forward Checking ergänzt, so dass ausschließlich zulässige Produktvarianten bei der paarweisen Kombination gebildet werden. Mit der hier vorgestellten Methode ist es möglich, eine vorliegenden Produktlinie des feasiPLe BMBF-Projekts [fC06] mit 1,8 Mio. Möglichkeiten auf 23 zu testende Produkte zu reduzieren.

Als Basis zur Erstellung einer repräsentativen Testmenge scheint das paarweise Testen vielversprechend zu sein. Exemplarische Studien unter Verwendung von pure::variants haben gezeigt, dass eine 100%ige Abdeckung aller paarweisen Kombinationen erreicht wird und Fehler in 3er-Kombinationen zu ca. 70-80% gefunden werden.

Weiterhin werden in zukünftigen Arbeiten Systemabhängigkeiten mit berücksichtigt, um unabhängige Teilbäume zu identifizieren, so dass nicht mehr alle Feature miteinander paarweise kombiniert werden, sondern nur noch voneinander abhängige. Ein erster Ansatz wurde in [OS09] erarbeitet und eine Evaluation ist Fokus aktueller Forschungsarbeiten. Zusätzlich werden in noch folgenden Beiträgen den Features Attribute zugewiesen, die den Kosten, Gewinn und Testaufwand beschreiben um die Auswahl der Varianten zusätzlich zu beinflussen. Features, die oft genutzt oder sicherheitskritisch sind, können so mit Priorität getestet werden. Vorteil dieses Ansatzes ist, dass Benutzungsszenarien und Anforderungen mit berücksichtigt werden können, die bestimmten Varianten einen höheren Stellenwert zuordnen als anderen Varianten [BC05].

Sollte sich in weiteren Arbeiten herausstellen, dass die Anwendung des kombinatorischen Testens auf SPLs nicht den erwarteten Erfolg bringt, so wurde in dieser Arbeit jedoch ein Algorithmus entwickelt der das paarweise Testen mit Berücksichtigung von Constraints und Forward Checking realisiert.

Literatur

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke und A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture - Foundations and Applications: Second European Conference*, Jgg. 4066 of *Lecture Notes in Computer Science (LNCS)*, Seiten 361–375, 2006.

[BC05] Renée C. Bryce und Charles J. Colbourn. Test prioritization for pairwise interaction coverage. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BG03] A. Bertolino und S. Gnesi. Use Case-based Testing of Product Lines. *SIGSOFT Software Engineering Notes*, 28(5):355–358, 2003.

[Bos00] Jan Bosch. *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley Longman, Amsterdam, 2000.

[BS04] J. Bach und P. Shroeder. Pairwise testing: A best practice that isn't. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*, Seiten 180–196. Citeseer, 2004.

[CDKP94] D. M. Cohen, S. R. Dalal, A. Kajla und G.C. Patton. The Automatic Efficient Tests Generator. *Fifth Int'l Symposium on Software Reliability Engineering*, IEEE:303–309, 1994.

[CDS07] M.B. Cohen, M.B. Dwyer und J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Intl. symp. on Software testing and analysis*, Seiten 129–139, 2007.

[CHE05] K. Czarnecki, S. Helsen und U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. Seiten 143–169, 2005.

[Con04] C. Condron. A Domain Approach to Test Automation of Product Lines. In *SPLiT 2004 Proceedings*, Seiten 27–35, 2004.

[CW07] K. Czarnecki und A. Wasowski. Feature diagrams and logics: There and back again. In *Software Product Line Conference*, Seiten 23–34, 2007.

[fC06] feasiPLE Consortium. www.feasiple.de. 2006.

[HE80] R.M. Haralick und G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[JhQJ08] L. Jin-hua, L. Qiong und L. Jing. The W-Model for Testing Software Product Lines. In *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, Jgg. 1, Seiten 690–693, 2008.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak und A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Bericht, Carnegie-Mellon U. SEI, 1990.

[LT98] Y. Lei und K.C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *IEEE High Assurance Systems Engineering Symposium*, Seiten 254–261, 1998.

[McG01] J. D. McGregor. Testing a Software Product Line. Bericht CMU/SEI-2001-TR-022, 2001.

[NFLTJ04] C. Nebut, F. Fleurey, Y. Le Traon und J.M. Jézéquel. A Requirement-Based Approach to Test Product Families. *LNCS, Springer-Verlag*, Seiten 198–210, 2004.

[Oli08] E. M. Olimpiew. *Model-Based Testing for Software Product Lines*. Dissertation, George Mason University, 2008.

[OS09] S. Oster und A. Schürr. Architekturgetriebenes Pairwise-Testing für Software-Produktlinien. In *Workshop Software Engineering 2009: PVLZ 2009*, March 2009.

[PBvdL05] K. Pohl, G. Boeckle und F. van der Linden. *Software Product Line Engineering*. Springer-Verlag, 2005.

[RKPR05] A. Reuys, E. Kamsties, K. Pohl und S. Reis. Model-based System Testing of Software Product Families. In *CAiSE*, Seiten 519–534, 2005.

[Sch07] Kathrin Scheidemann. Verifying Families of System Configurations. *Doctoral Thesis*, TU Munich, 2007.

[SM98] B. Stevens und E. Mendelsohn. Efficient software testing protocols. In *Conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1998.

[TTK04] A. Tevanlinna, J. Taina und R. Kauppinen. Product Family Testing: a Survey. *ACM SIGSOFT Software Engineering Notes.*, 29, 2004.

[WSS08] S. Weißleder, D. Sokenou und B. Schlinglo. Reusing State Machines for Automatic Test Generation in Product Lines. In *MoTiP Workshop*, 2008.

The Impact of Variability Mechanisms on Sustainable Product Line Code Evolution

Thomas Patzke

Product Line Architectures Department
Fraunhofer Institute Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern
thomas.patzke@iese.fraunhofer.de

Abstract: Many software development organizations today aim at reducing their development effort, while improving the quality and diversity of their products by building more reusable software, for example using the product line approach. A product line infrastructure is set up for deriving the similar products, but this infrastructure degenerates over time, making reuse increasingly hard. As a countermeasure, we developed a practical method for guiding product line developers in evolving product line code so that its decay caused by reuse is avoided. This paper gives an overview of some of our findings.

Because product line code differs from single systems code only in its genericity, expressed by variability mechanisms, we analyzed to what degree the selection of certain mechanisms affect the code's reuse complexity. Using the Goal-Question-Metric (GQM) approach, we developed a quality model that lead to an extensible product line complexity metrics suite.

A case study compared the evolution qualities of different product line implementations, with the following results: Cloning, the simplest mechanism, leads to similar short-term complexities than more advanced ones, making its interim usage appropriate. In the longer term, any other mechanism has a clearly lower complexity trend, especially if it is selected according to the variability management task at hand. A mix of Conditional Compilation and Frame Technology provides the best long-term evolution potential.

1 Introduction

Many software development organizations today aim at reducing their development effort, while at the same time improving the quality and diversity of their software products by building more reusable software. Due to the diversity of similar software products, it is often insufficient to provide reusable software as fixed building blocks only. Truly reusable software requires adaptation possibilities in order to be reused effectively in different situations. This means that although it contains common parts, it must also provide some means to variation. More recently, more and more reusable

software of larger scale is developed in a conscious organization-specific engineering effort alongside the software that reuses it. In these approaches, a set of similar software systems is developed together in a cost-effective way. In these product line engineering approaches, the reusable assets consist of common and variable artifacts, plus descriptions of variability interdependencies, across all stages of the software engineering life cycle, such as requirements, architecture, design or source code. Together they form the organization's product line infrastructure [Mut02].

Once a product line infrastructure has been initiated, individual software products can be derived from it, and if the infrastructure is well-designed initially, it should be easy to derive products from it for some time. However, development in general and product line development in particular do not end with initial construction, but successful development is accompanied by continuous evolution ([Par94], [Som04], [LF06]). In a product line setting, continuous evolution means that gradually, new products are added to the product line and existing common and variable features are changed which were only partially predicted, or which were not predicted at all.

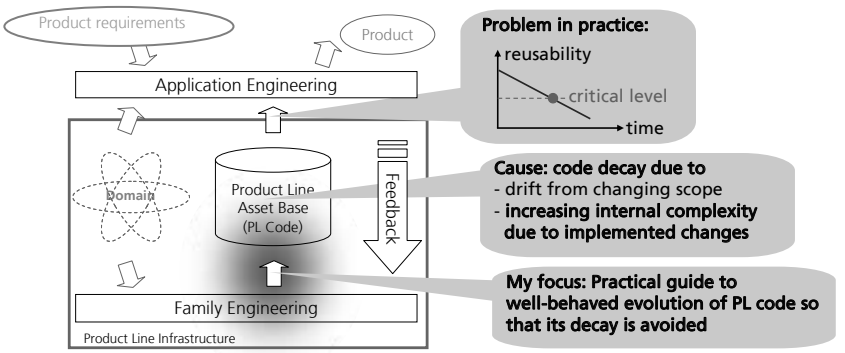


Figure 1: The product line code evolution problem

Figure 1 shows our observations in practice: Within the product line engineering life cycle, it becomes increasingly difficult to derive products from the product line code over time. The reason for this phenomenon is product line aging, analogous to single systems software aging [Par94]. Code degenerates for two reasons: First, because it is not changed according to the changing scope, so that both drift apart. Second, it ages when it is changed inappropriately [Kru07], for example because the developers are ignorant of product line implementation technology or because they adopt it overambitiously. Our method focuses on the latter issue. It analyzes practical possibilities for implementing variabilities in source code and instantiates a sustainable coding process.

As shown in Figure 2, existing product line code and new product line specifications are the input to our approach, targeting the developer. Other inputs are variability mechanisms (possibilities to realize the new variability-related specifications in the code; Chapter 3) and product line evolution scenarios (elementary ways in which variabilities may evolve). The output of the iterative and incremental approach is new product line

code with just enough complexity to be easily evolvable. The iteration consists of selection, modification (refactoring) and quality assurance phases. Quality assurance consists of product line testing and measurement (Chapter 4) sub-activities which ensure that all product line members can be constructed and executed as required, and that the applied variability mechanisms are simple enough for sustainable evolution.

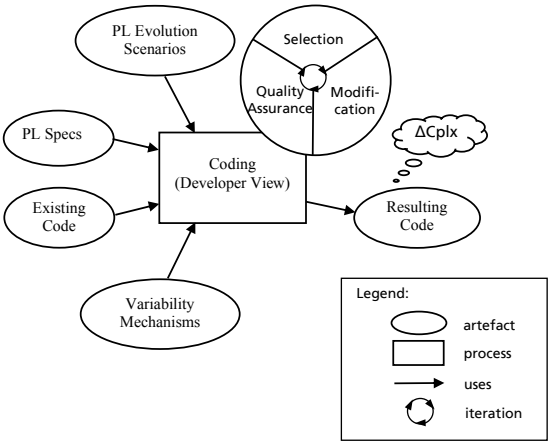


Figure 2: Product line coding approach

The method was developed and applied in a case study (Chapter 5), in which product lines of wireless sensor node software have been developed. Chapter 6 gives a summary and presents open issues and further work.

2 Related Work

Our approach for combines and extends theories and concepts from several different areas, such as software evolution, product line implementation, practical reuse methods, complex systems description and sustainable evolution of complex systems.

Observing the long-term evolution of large single systems used in practice, 8 evolution rules have been proposed [LF06], including the phenomenon of software aging [Par94], and a number of complexity measurements been proposed ([AD07], [Kel06]). Collections of product line implementation mechanisms ([AG01], [SVB05], [CE00], [JGJ97]) have mostly had a theoretical scope, sometimes biased towards a particular solution, but not regarding long-term effects. More practical methods have been proposed ([MP03], [Bas97], [Kru07], [Cop98]), which only partially address sustainable evolution. Work on complex systems description ([Sim62], [Ley01], [CZ03]) suggested that such systems should be described by actions leading to symmetries, rather than just describing the shape of end products. Whereas these are only analytical, work on sustainable evolution of such systems [Ale02] suggests a generic recipe for actively creating well-behaved complex systems, involving certain steps in a particular order. We developed our method as an instance of that approach.

3 Variability Mechanisms

As mentioned in Chapter 1, the additional complexities in product line code evolution compared to single systems code evolution have to do with variability and how variability is managed in the code. A developer may select from a number of different variability mechanisms to realize and configure a new product instance that reuses a given product line infrastructure. In practice, variability mechanisms are often used ad-hoc or in monocultures, which leads to overly complex code. In order to guide the developer in selecting appropriate mechanisms, we have developed a pattern language of variability mechanisms [Pat08], summarized in this chapter. The selected mechanisms include Cloning, Conditional Execution, Polymorphism, Partial Binding, Conditional Compilation, Aspect-Orientation and Frame Technology. We believe that our pattern language captures the major possibilities for organizing variable code for two reasons. First, because each chosen mechanism is frequently used for variability management in practice or has shown new and unique variability management possibilities in practical research. Second, because from a developer's perspective, the mechanism set covers all major variability management situations, as illustrated in Figure 3.

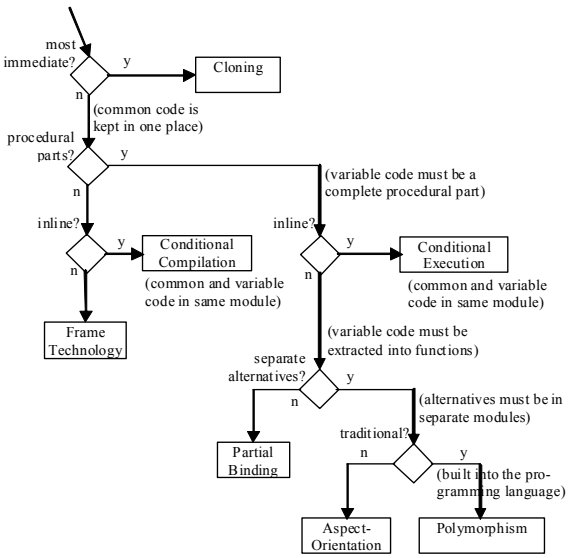


Figure 3: Variability management decisions resulting in variability mechanisms

When a new product must extend existing infrastructure code, the simplest approach is **Cloning**, i.e. to duplicate the existing code and to modify the new copy. Cloning is simple because at the moment when it is applied it achieves just what is required in the most direct way: the new product is based on existing code, and existing products are not affected. However, the approach is least sustainable, because it leads to maximum duplication of common code, which must be co-evolved consistently throughout the product line's future lifetime, which is most of the time the code spends its life. This way, Cloning it is most detrimental for long-term product line code evolution.

The only sustainable alternative is to sacrifice direct duplication, so that at least some common code is kept in one place, while the new variability is introduced otherwise. A straightforward way for differentiating a variable code part from previously undifferentiated code is to enclose it by a conditional statement. The new variability is added as a new branch of the conditional statement. The resulting variability is bound at execution time, which is why the mechanism is called **Conditional Execution**. A disadvantage of this approach is that common and variable code parts reside and evolve in a single module, because the parameter of variation is closed [Bas97]. This problem is solved in many programming languages by using an indirect rather than a direct conditional statement for differentiating the variable parts, for example through function pointers, Template Methods, or generic types [CE00]; that is, by **Polymorphism**.

If the variation points can be identified as unique procedures in the common code, a non-traditional way is to extract the variable parts into aspects, using **Aspect-Orientation**. A more traditional alternative is to store all variable procedural code parts in a single module, different from the common code. We call this mechanism **Partial Binding**, because the parameters of variation are partially bound: their calling sites do not specify completely which functions they call. For example, they only provide a forward declaration, and the missing function definition is completed by stating the variable module at preprocessing- or link-time.

Except for Cloning, the above mechanisms are programming language-dependent. Conditional Execution and (runtime) Polymorphism are especially problematic in the larger context of product line implementation because they result in runtime binding, which leads to a single bloated system that contains all variable features, rather than a proper product line, a set of similar systems evolved together. Runtime conditionals are also inherent parts of software code, and if they are also used for variability management, “normal” code and variability managing code become deeply entangled. However, it is not necessary to express the condition with programming language code. An alternative is the preprocessor conditional, such as the #ifdef statement of the C and C++ preprocessor, binding the parameters of variation at preprocessing time, in **Conditional Compilation**. This way, variable parts may cover arbitrary code sections, independent of their semantics. Conditional Compilation facilitates reuse in a broader sense, where the preprocessor serves as a construction interpreter [Bas97]; like Conditional Execution it is a closed-parameter mechanism. At least one open-parameter construction-time mechanism exists: **Frame Technology** [Bas97], which uses an advanced preprocessor, such as the FP frame processor developed by the author of this paper [PM03], for assembling construction modules. These may contain source code text, whose default variable parts are annotated and can be overridden.

4 Product Line Complexity Measurement

As mentioned in Chapter 1, one sub-activity of the quality assurance phase of our method is concerned with measurement in the resulting product line code (cf. Fig. 2), more precisely with complexity measurement. Until recently, there had been no such research for product lines [Kna04]; this situation is currently changing ([AD07],

[LT08]). However, none of these consider the measurement purpose. We developed a quality model for evolving product line code that motivates which product line-specific quality attributes must be addressed in the measurement phase. It is customizable to an organization's product line development needs. The quality model has been developed using the Goal-Question-Metric (GQM) approach [S++02], the most popular mechanism for goal-oriented software measurement. GQM depends on the formulation of the following elements: Goals, questions and metrics. First, goals are formulated, which define what shall be achieved. The goals are often decomposed using a goal hierarchy of main goals and corresponding sub-goals. Each goal is refined by questions, whose answers indicate to what extent the goal has been reached. Finally, metrics are given for each question, which makes the questions quantifiable. Figure 4 shows the goals and sub-goals needed in our method.

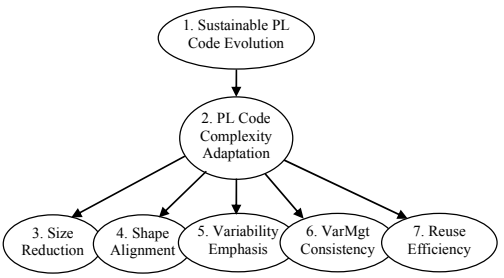


Figure 4: Goal hierarchy of the product line code quality model

The main goal addressed is sustainable evolution of product line code (G1), which aims at making the product line code evolvable over longer periods of time, possibly open-ended. In order to achieve the main goal, product line complexity adaptation is proposed as a sub-goal (G2), which denotes that product line code complexity is reduced, but only as far as necessary in the particular development context. For example, product line code complexity is too high if a module contains more parameters of variation than necessary. There are five mostly orthogonal sub-goals for achieving complexity adaptation. The first is size reduction (G3), which means that the product line code is constrained in size, as required for the variability management purposes at hand. The second sub-goal, code shape alignment (G4), denotes that the shape of common and variable code parts corresponds to the required variability management tasks. For example, closed-parameter mechanisms are usually sufficient for expressing optional variabilities; open-parameter mechanisms would introduce gratuitous complexity. The third sub-goal is concerned with emphasizing variable parts (G5), so that developers can easily see and control those parts of product line code which are different across space or time, while at the same time the common parts are more subdued [Bas97]. Another sub-goal addresses variability management consistency (G6), because inconsistent code is more complex to evolve than necessary, and the last sub-goal is reuse efficiency (G7), which means, for example, that an excess of reusable modules may become as harmful for long-term reuse as a shortage of reusable modules.

According to the GQM method, each goal is refined by questions, and these are then refined by concrete metrics. Table 1 gives an example for the size reduction goal G3. This way, we established a metrics suite of 21 metrics.

Analyze the	code of software product lines
for the purpose of	**reducing**
with respect to	**size**
from the viewpoint of the	software developer

Q7: How large is the code? (Which code size is necessary?)
Q8: How much product line code has changed over time? (How much was necessary?
Q9: How many modules have been used? (How many are necessary?)
Q10: How many variation points are used in the code? (How many are necessary?)

G	Q	Metric name	Description
3	Size reduction		
	7	LOC	Lines of code for entire product line code
	8	$\nabla_{LOC,t}$	Temporal code churn ([HM00], [AD07]) in lines of code
	9	NOM	Number of modules
	10	NVP	Number of variation points

Table 1: Metrics for goal G3: Product line code size reduction

5 Case Study

In order to compare the quality of sustainable code evolution in various product line development contexts, we conducted a case study that monitored and evaluated the evolution of software product lines of small and highly resource-constrained embedded systems. The product line implementations were co-evolved by using each of the mechanisms presented in Chapter 3, applying the overall method shown in Figure 2. In particular, the quality of the results was measured using our quality model (Chapter 4). In this context, four main hypotheses have been investigated, summarized in Table 2.

No	Hypothesis
1	Except in the short term, code obtained by Cloning is harder to evolve than code with any other variability mechanism.
2	In the long term, a monoculture of a variability mechanism is harmful for product line code quality.
3	Runtime variability mechanisms unnecessarily increase product line code complexity.
4	As a variability mechanism, Aspect-Orientation is obsolete.

Table 2: Overview of investigated hypotheses

The single systems case, when no variability management is applied, corresponds to the situation when Cloning is used throughout. This results in the first hypothesis, which claims that Cloning has a comparable complexity than the other mechanisms only in initial product line development phases, while it has a more than linear complexity trend compared to other mechanisms in the mid- and long term. The other hypotheses are concerned with groups of mechanisms, except for Cloning. Hypothesis two states that, in

the long term, a monoculture of one variability mechanism, for example only Conditional Compilation, as observed in practice, leads to less sustainable code than a context-specific mix of mechanisms. The third hypothesis states that in most cases, runtime variability mechanisms make the code unnecessarily complex, so that the dual construction time mechanisms should be used instead. The fourth hypothesis claims that Aspect-Orientation is obsolete for variability management, as other mechanisms lead to the same result with less effort and complexity.

The subject of the case study is code for small embedded systems - battery-powered wireless sensor nodes which are part of a rapid prototyping platform for Ubiquitous and Pervasive Computing environments [TeCo]. The sensor nodes are able to communicate with internet gateways or with each other, forming an ad-hoc wireless sensor network. They are equipped with various types of actuators and sensors. The case study simulates the evolution of a product line in six steps, as shown in the feature diagram in Figure 5.

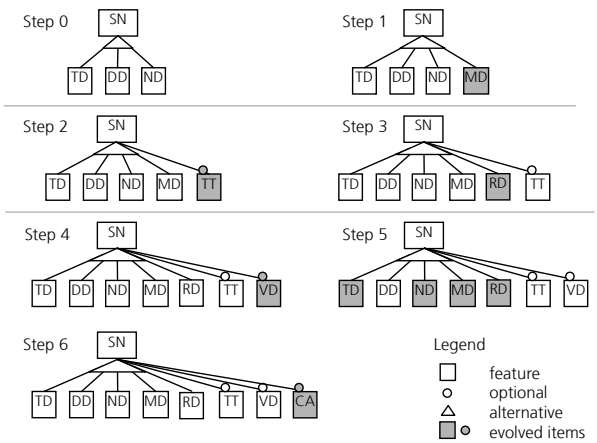


Figure 5: Feature diagram snapshots of the evolving sensor node product line

Initially, the product line consists of the sensor node (SN) and the three alternative features tilt detector (TD), drop detector (DD) and noise detector (ND). In step 1, a new alternative movement detector (MD) is added, step 2 adds an optional time transmission (TT), step 3 a raw data detector (RD), and step 4 a voltage detector (VD). In step 5, most existing alternatives are simplified, and in step 6 a clock adjustment feature (CA) is added.

Figure 6 depicts the evolution trace of the product line code. The sequences “a” to “g” have each been coded using a monoculture of the seven variability mechanisms discussed in Chapter 3. The respective implementations at $t=t_0$ serve as a temporal baseline, to evaluate how complexity changed over time. Sequence “i” represents the “ideal” implementation at each point in time, serving as a spatial baseline, to evaluate how much more complex each implementation is than required. It contains the same C code as in the other implementations, enriched with variability management pseudocode. The pseudocode expresses what activities a human developer or an automated

construction interpreter must at least perform in order to assemble all required product instances from the C code parts, e.g. by changing the text at certain lines. It separates what must be performed for "good enough" variability management from how to achieve these tasks. The remaining sequence "h" uses a mix of the available variability mechanisms, staying as close as possible to the baseline "i".

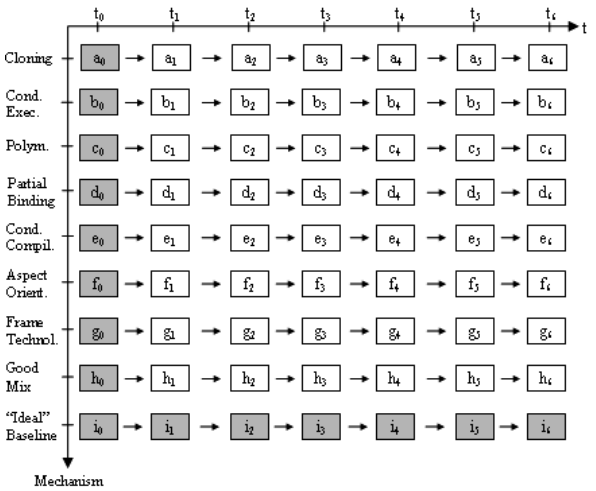


Figure 6: Evolution trace for product line code, with baselines (grayed)

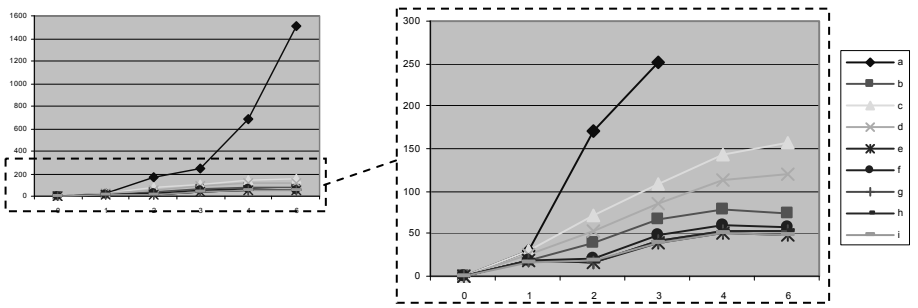


Figure 7: Trends for temporal code size delta (left), and excerpt (right)

While keeping all product line implementations consistent (invariant goal G6), trends for the 17 metrics of all remaining goals were captured, partially automated. As an example, Figure 7 shows the trends for temporal code churn, corresponding to question Q8 in Table 1. The values were then normalized against the worst figures except for Cloning, because its metrics often exceeded the others' considerably. The values ranged between zero for the ideal implementation "i", and one for the worst case. The corresponding values are listed in Table 3. Thereafter, average values were computed for the corresponding goal categories. The same process was repeated for all seven evolution scenarios. Figure 8 summarizes the results.

	LOC	$\nabla_{LOC,t}$	NOM	DRH	WRH	$v(G)_{t,closed}$	$v(G)_{t,open}$	$v(G)_{ct,closed}$	$v(G)_{ct,open}$	LOC_{ad}	$NVPrt_e$	$NVPrt_i$	$NVPrt_a$	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	9,63	11,8	3,18	0,3	5,1	19	2	4	1	12,6	1	1	0	1	1	10	0,3
b	0,23	0,17	0,36	0,3	0,4	0,8	0	0,9	0	0,85	1	1	1	1	1	0,5	1
c	1	1	0,73	0,3	0,6	1	1	0,6	0	0,99	1	1	0,67	0,39	1	0,6	0,18
d	0,68	0,65	1	0,3	1	0,4	0	1	1	1	1	1	0	0,62	1	1	0,17
e	0,05	0,03	0,36	0,3	0,4	0	0	0,4	0	0,62	1	1	0	1	1	0,5	1
f	0,12	0,11	0,27	0,3	0,4	0,5	0	0	0	0,27	1	1	0,5	0,33	1	0,1	0,03
g	0,1	0,09	0,27	1	0	0,4	0	0	0	0,27	1	0	0	0,36	0	0,1	0,05
h	0,1	0,05	0	0	0	0,2	0	0	0	0,06	0	0	0	0,03	0	0,1	0,02
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Name	Meaning
LOC	lines of code
$\nabla_{LOC,t}$	temporal code churn
NOM	number of modules
NVP	number of variation points
DRH	depth of reuse hierarchy
WRH	width of reuse hierarchy
v(G)	cyclomatic complexities
LOC_{ad}	lines of adaptee code
NVPrt	number of variable parts
RR	reuse ratio
NOD	number of defaults
$\nabla_{LOC,s}$	spatial code churn
K_{var}	compression distance of alt. var.

Table 3: Normalized metrics for all mechanisms after evolution step 6

cplx	0	1	2	3	4	5	6
a	0,55	0,64	1,26	1,42	2,65	2,5	4,57
b	0,77	0,77	0,74	0,83	0,74	0,73	0,67
c	0,65	0,61	0,75	0,75	0,74	0,74	0,75
d	0,57	0,52	0,68	0,68	0,7	0,7	0,72
e	0,67	0,69	0,61	0,66	0,56	0,53	0,48
f	0,13	0,11	0,3	0,33	0,35	0,36	0,37
g	0,06	0,06	0,14	0,14	0,19	0,2	0,24
h	0,06	0,06	0,06	0,06	0,05	0,06	0,05
i	0	0	0	0	0	0	0

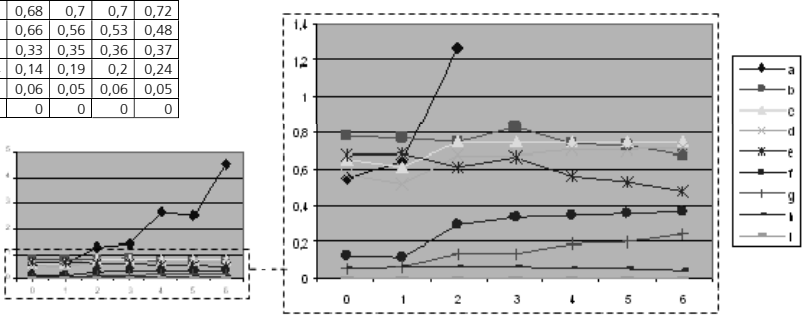


Figure 8: Total complexity trends, compared to ideal implementation

The case study shows that after few product line evolution scenarios, Cloning leads to a considerable excess of complexity, supporting hypothesis 1. The three conventional variability mechanisms Conditional Execution, Polymorphism and Partial Binding result in comparable levels of complexity, with weaknesses in different dimensions. The runtime mechanism Conditional Execution results in 42% higher average complexity than its construction time dual Conditional Compilation, especially due to lack of variability emphasis. This clearly supports part of hypothesis 3. On the other hand, the second runtime mechanism, Polymorphism, only performs 4% worse than its dual, Partial Binding, which does not support hypothesis 3. Conditional Compilation leads to some variability management complexity, mainly due to missing optimization possibilities, but scores best among the closed-parameter mechanisms. Aspect-Orientation performs better, but makes defaults and variation points hard to see. Frame Technology, even in the restricted dialect used in the case study that only supports open parameters of variation, clearly outperforms Aspect-Orientation in almost any category. For that reason, Aspect-Orientation is obsolete when both mechanisms are available, supporting hypothesis 4. Frame Technology performs best among the mechanism monocultures, with a 24% deviation from the ideal implementation after the final evolution step. However, it over-emphasizes variable parts in some cases and leads to a lack of open construction time complexity, due to limitations of the applied frame processor. All monocultures lead to complexity excess, which supports hypothesis 2.

The approach has the following threats to validity: First, the case study has deliberately been kept small, and it can be argued if the found results scale to larger product line code. However, practical experience in various projects indicates that the given hypotheses are useful heuristics for a majority of product line development projects. Second, it may be argued that the chosen scenarios in the case study are non-representative for typical product line implementation problems, for example because they are too few, too simple, too orthogonal, or because they leave out numeric variabilities. While variabilities of the numeric type occur occasionally, they typically do not pose a serious coding problem, and especially, they do not rely on the presence of most discussed variability mechanisms, which is why they have not been covered in the case study. Another threat of validity is that the chosen reference system is biased towards certain mechanisms or mechanism combinations, and that the underlying heuristics for representing alternative and optional variabilities do not hold. While it is true that representation of variabilities in code is situation-dependent, the heuristics acknowledge this fact. On the other hand, if considering complexity in terms of necessity and arbitrariness, there is strong evidence that many optionals do not require as open parameters of variation as alternatives do, which is why open-parameter mechanisms have been used in the reference implementation for the former, while the latter have preferred closed-parameter mechanisms. A final threat to validity is if an appropriate set of goals and metrics has been used, and if the smaller measured differences among many mechanisms beyond Cloning could rather mean that any mechanism may be applied on equal terms. While there are indicators in single systems development that many metrics strongly correlate with code size, lines of code cannot be the only indicator for product line code quality, because not all variability management and evolution problems are solved by just maximally collapsing the code. There must be other factors for distinguishing more suitable variability mechanisms from less suitable ones, because otherwise, for example, it would not make a difference if Conditional Execution or Conditional Compilation is used, which often differ hardly in terms of code size. Here, the differences are more in code shape, or in the visibility of variable parts, which help or prevent the developer from keeping product line code reusable.

6 Summary and Outlook

This paper presented an overview of our approach for keeping product line code reusable in practice. Because product line code becomes more complex than single systems code due to its additional variability, we analyzed the impact of major variability mechanisms on evolvability. A product line quality model was developed according to the GQM approach, and the proposed metrics were applied in a case study. The results show that Cloning can be appropriate in the short term, while a mix of Conditional Compilation and Frame Technology yield the best long-term results.

Further work will extend the approach to single systems development, evaluating which semantic source code elements are essential and which cause arbitrary complexities in specific contexts, in order to deliberately omit the latter elements when there is no reason to use them. A second extension is to broaden approach from coding activities to other software development activities, for example architecting, so that all product line

infrastructure artifacts are systematically simplified and become more evolvable. Third, the scalability of the approach will be evaluated by using large-scale code. Fourth, synergies will be evaluated with other product line implementation technologies, especially configuration management.

References

[AD07] S.A. Ajila, R.T. Dumitrrescu: Experimental Use of Code Delta, Code Churn, and Rate of Change to Understand Software Product Line Evolution. JSS 80(1), 74–91, January 2007

[Ale02] C. Alexander: The Nature of Order, Book 2. CES Publishing, 2002

[AG01 M. Anastasopoulos, C. Gacek: Implementing Product Line Variabilities. ACM Software Engineering Notes 26(3): 109-117, May 2001]

[Bas97] P. Bassett: Framing Software Reuse. Lessons From The Real World. Prentice Hall, 1997

[CE00] C. Czarnecki, U.W. Eisenecker: Generative Programming. Addison-Wesley, 2000

[Cop98] J.O. Coplien: Multi-Paradigm Design for C++. Addison-Wesley, 1998

[CZ03] J.O. Coplien, L. Zhao: Symmetry and Symmetry Breaking in Software Patterns. GCSE-2: 37-56 (LNCS2177), Springer-Verlag, 2000

[HM00] G.A. Hall, J.C. Munson: Software Evolution: Code Delta and Code Churn. JSS 54(2): 111-118, October 2000

[JGJ97] I. Jacobson, M. Griss, P. Jonsson: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, 1997

[Kel06] D. Kelly: A Study of Design Characteristics in Evolving Software Using Stability as a Criterion. IEEE Transactions on Software Engineering 32(5): 315-329, May 2006

[Kna04] P. Knauber: Managing the Evolution of Software Product Lines. ICSR-8, 2004

[Kru07] C.W. Krueger: The 3-Tiered Methodology: Pragmatic Insights from New Generation Software Product Lines. Proc. SPLC-11: 97-106, 2007

[Ley01] M. Leyton: A Generative Theory of Shape (LNCS 2145). Springer-Verlag, 2001

[LF06] M.M. Lehmann, J.C. Fernandez-Ramil: Rules and Tools for Software Evolution Planning and Management. In N.M. Madhavij, J. Fernandez-Ramil, E.E. Perry (Eds.): Software Evolution and Feedback - Theory and Practice. Wiley & Sons, 2006: 539-563

[LT08] R.E. Lopez-Herrejon, S. Trujillo: How complex is my Product Line? The case for Variation Point Metrics. VAMOS08 Workshop, 2008

[Mut02] D. Muthig: A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines. PhD Thesis, Fraunhofer-Verlag, 2002.

[MP03] D. Muthig, T. Patzke: Generic Implementation of Product Line Components. NetObjectDays 2002: 313-329

[Par94] D.L. Parnas: Software Aging. ICSE-16: 279-287, 1994

[Pat08] T. Patzke: Mechanisms, Processes and Metrics for Implementing and Evolving Reusable Embedded Systems. Fraunhofer IESE Report 021.08/E, 2008

[PM03] T. Patzke, D. Muthig: Product Line Implementation with Frame Technology: A Case Study. Fraunhofer IESE Report 018.03/E, 2003

[S++02] R. van Solingen, V.R. Basili, G. Caldiera, H.D. Rombach: Goal GQM Approach. In J.J. Marciniak (Ed.): Encycl. Software Engineering (2nd Ed.), Wiley & Sons, 2002: 578-583

[Sim62] H.A. Simon: The Architecture of Complexity. In H.A. Simon (Ed.): The Sciences of the Artificial (3rd ed.). MIT Press, 1996

[Som04] I. Sommerville: Software Engineering (7th Ed.). Pearson Education, 2004

[SVB05] M. Svahnberg, J: van Gurp, J. Bosch: A Taxonomy of Variability Realization Techniques. Software - Practice and Experience 35: 705-754, Wiley & Sons, 2005

[TeCo] Particle Computer homepage: particle.teco.edu (Retrieved October 2009)

Entwicklung eines objektiven Bewertungsverfahrens für Softwarearchitekturen im Bereich Fahrerassistenz

D. Ahrens[1], A. Frey[1], A. Pfeiffer[1] und T. Bertram[2]

[1]Fahrdynamik
Fahrerassistenz und aktive Sicherheit
BMW Group
80788 München
dirk.ahrens@bmw.de

[2]Lehrstuhl für Regelungssystemtechnik
Technische Universität Dortmund
44221 Dortmund
torsten.bertram@tu-dortmund.de

Abstract: Der vorliegende Beitrag beschreibt ein Vorgehensmodell und die erzielten Ergebnisse im Umfeld der Bewertung von Softwarearchitekturen von Automotive Embedded Software. Softwarearchitektur stellt im Allgemeinen ein entscheidendes Instrument zur Beeinflussung nicht-funktionaler Eigenschaften (z.B. *Skalierbarkeit*, *Erweiterbarkeit*, *Portierbarkeit*) von Software dar, bis heute gibt es jedoch keinen geeigneten Ansatz, die Qualität solcher Architekturen *objektiv* und *quantitativ* zu ermitteln und zu bewerten.

Zunächst wird ein *Qualitätsmodell* mit einer Vielzahl unterschiedlicher Kriterien aufgestellt, welches auf die Bedürfnisse von Automotive Embedded Software angepasst ist. Für die relevanten Kriterien werden auf dieser Basis insgesamt *acht Metriken* zur quantitativen Beurteilung von Automotive Softwarearchitekturen erarbeitet.

Zur automatisierten Anwendung dieser Metriken ist eine Einbettung in den aktuellen Entwicklungsprozess notwendig. Diese Anpassungen und die daraus resultierende und eingesetzte Infrastruktur für den Bewertungsablauf wird ebenso vorgestellt wie der schlussendlich *implementierte Softwareprototyp* auf Java-Basis. Mit diesem Entwicklungswerkzeug ist eine automatisierte, schnelle, komfortable und individuell konfigurierbare Bewertung von vorhandenen Softwarearchitekturentwürfen möglich.

Abgerundet wird der Beitrag durch ausgewählte Anwendungsbeispiele und den Ausblick auf die weiteren anstehenden Arbeiten in diesem Umfeld.

1 Einführung und Problemstellung

Die Entwicklung von Automotive Embedded Software steht momentan vor großen Herausforderungen. Einerseits ist sie längst als wichtigster Schlüssel und Motor für kommende wettbewerbsentscheidende Innovationen identifiziert, andererseits muss die Entwicklung, Umsetzung und Absicherung in immer kürzeren Zeitabständen unter Einsatz von knapper werdenden (menschlichen und finanziellen) Ressourcen für gleichzeitig immer mehr Baureihen, Fahrzeugtypen und Derivate (kurz: Varianten) geschehen. Es gilt somit zwangsläufig einen unvermeidbaren Konflikt zwischen steigender Funktionalität, Komplexität sowie Vernetzung und wesentlichen nicht-funktionalen Anforderungen wie *Skalierbarkeit*, *Erweiterbarkeit*, *Portierbarkeit* und *effizienter Ressourcenausnutzung* so gut wie möglich zu lösen. Durch den Entwurf einer Softwarearchitektur, welcher üblicherweise sehr früh (s. Abb. 1 rechts) im Entwicklungsprozess entsteht, werden die obigen Eigenschaften der Software entscheidend – positiv wie negativ – beeinflusst. Nachträgliche Änderungen an der Architektur sind zu einem späteren Zeitpunkt nicht mehr oder nur durch enormen Aufwand durchführbar. Einem guten und reifen Architekturentwurf vor der Implementierung kommt somit eine entscheidende Bedeutung zu.

In der heute gängigen Praxis bei der Entwicklung von Softwarearchitekturen stellt sich erst am Ende der Entwicklung heraus, ob und wie gut die nicht-funktionalen Anforderungen von der Software(-architektur) erreicht werden. Unzulänglichkeiten können dann kaum noch korrigiert werden. Alle Entwurfsentscheidungen hängen stark von bewährten Mustern (allgemeinen oder firmenspezifischen) ab und sind auf subjektive und intuitive Entscheidungen von Experten und deren persönlicher Erfahrung angewiesen. Dieses individuelle Wissen ist in den Köpfen weniger Mitarbeiter gebunden und kann nur schwer dokumentiert oder an neue Mitarbeiter weiter gegeben werden. Sind neue Personen an der Entwicklung beteiligt, muss also aufgrund mangelnder Erfahrung bei „Null“ angefangen werden. Gleichzeitig bedeutet die Tatsache, dass gewisse Muster oder Praktiken seit längerer Zeit angewendet werden, nicht automatisch, dass diese auch wirklich gut in Hinblick auf die spezifischen Anforderungen sind. Es bleibt unklar, wie viel Potential zu Verbesserungen noch besteht. Gänzlich fehlend sind also Möglichkeiten die „Güte“ einer Softwarearchitektur *zuverlässig*, *reproduzierbar* und vor allem *objektiv* feststellen und mit anderen Entwürfen vergleichen zu können, um dadurch eine zielgerichtete Weiterentwicklung und iterative Evolution der Architektur zu ermöglichen.

Ziel der vorliegenden Arbeit ist es einen Ansatz zu entwickeln, mit dem geeignete Kriterien zur Softwarearchitektur-Bewertung gefunden, definiert, formalisiert und *quantifiziert messbar* gemacht werden können, um im Anschluss als *objektive Metriken* automatisiert auf eine in einem definierten, noch festzulegenden Datenformat gegebene Softwarearchitektur angewendet zu werden.

Bisherige Ansätze [Bo78][ED96][ISO01][Oe06] beschäftigen sich bisher ausschließlich mit der Bewertung von bereits implementierter Software und nicht mit den im Entwicklungsprozess deutlich früher zu entwerfenden Architekturen oder sie beschäftigen sich zu oberfläch mit den Möglichkeiten zur Bewertung von Softwarearchitekturen. Es werden meist nur allgemeine Kriterien, Empfehlungen und Ideen geschildert ohne objektivierte oder formalisierte Hilfsmittel anzubieten [Ab96][EHM08][Lo03][PBG07][RH06].

Der hier vorgestellte Ansatz schließt diese Lücke und bietet dem Anwender erstmals ein echtes Hilfsmittel zur *objektiven und quantifizierten Bewertung von Softwarearchitekturen*, welches einen entscheidenden Beitrag zur Verbesserung von Softwarequalität bezüglich nicht-funktionaler Anforderungen leisten kann. Dazu werden die Kriterien aus den bestehenden Ansätzen gesammelt, kombiniert und an die speziellen Bedürfnisse der Automotive Softwareentwicklung am Beispiel von Fahrerassistenzsystemen angepasst. Für das daraus entstandene *Qualitätsmodell* wird untersucht, welche Kriterien besonders relevant für die Domäne und welche davon überhaupt quantitativ erfassbar sind. Die erarbeiteten Metriken werden gesamthaft und an einigen Beispielen auch im Detail vorgestellt. Darüber hinaus wird zum besseren Verständnis ebenfalls der *umgebende Entwicklungsprozess* als notwendige Infrastruktur sowie der implementierte *Software-Prototyp*, welcher eine automatisierte Anwendung der Metriken auf gegebene Softwarearchitekturen ermöglicht, präsentiert. Aktuelle Erkenntnisse aus der Anwendung der gesamten Wirkkette sowie der Ausblick auf weitere Arbeiten runden den Beitrag ab.

2 Einbettung in den Entwicklungsprozess

Wichtig zum weiteren Verständnis des Bewertungsverfahrens ist die Kenntnis der Randbedingungen zu seinem Einsatz. Es wurde im Vorfeld ein Entwicklungsprozess (Abb. 1) ausgearbeitet, welcher eine hohe Durchgängigkeit, Formalismen sowie automatisierte (teil- und vollautomatisierte) Übergänge zwischen Prozessschritten ermöglicht und somit als Ergänzung beziehungsweise Erweiterung der bisher in der Praxis üblichen Entwicklungsschritte eingesetzt werden kann und soll.

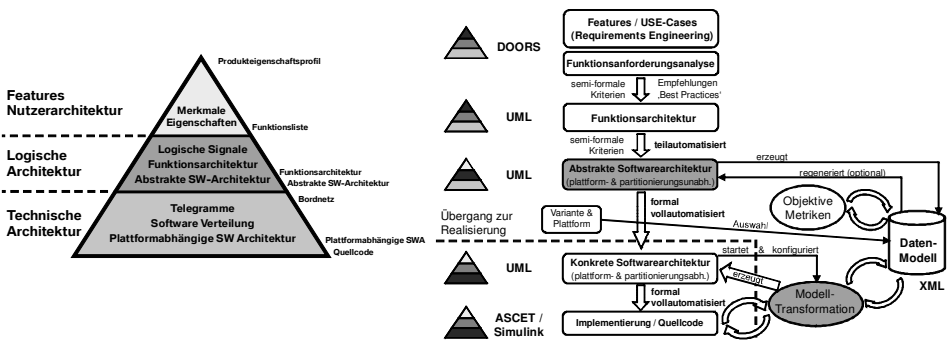


Abbildung 1: Rahmen bildendender Softwareentwicklungsprozess

Details zum umgebenden Prozessrahmen (Abb. 1 links), den detaillierten Prozessschritten (Abb. 1 rechts) sowie den entwickelten Konzepten und Übergängen können [Ah08] [Ah10][APB08] entnommen werden. Für das Verständnis der vorliegenden Arbeit wesentlich sind die auf der rechten Seite der Abbildung hervorgehobenen beiden Prozessschritte. Dabei handelt es sich zum einen um die abstrakte Softwarearchitektur, eine auf einem UML Metamodell [OMG05] basierende Darstellung speziell erstellt für Automotive Softwarearchitekturen [Ah08][Ah09][APB08] und zum anderen um das Konzept der automatisierten Modelltransformation, welche die bidirektionale Überführung zwischen der UML-Darstellung und den Implementierungswerkzeugen ermöglicht.

Zu diesem Zweck wird als Zwischenschritt ein formales konsistenzwahrendes Datenmodell auf XML-Basis [W3C98], beschrieben durch ein XML-Schema, verwendet.

Die Anwendung der Metriken selbst soll auf Basis dieser Datenbank erfolgen, so dass sowohl ein Top-Down-Vorgehen für in der UML erstellte Entwürfe als auch ein Bottom-Up-Vorgehen zur Bewertung und Neustrukturierung bereits bestehender Architekturen (kommend von den Implementierungs-Tools) möglich ist. Details zum Datenmodell sowie der konkreten Umsetzung und Anwendung der Metriken finden sich in Kapitel 4.

3 Qualitätsmodell und die erarbeiteten Bewertungsmetriken

Bisher wurde der Begriff „Softwarearchitektur“ bereits mehrfach verwendet. Da sowohl in der Literatur als auch im fachlichen Sprachgebrauch keine allgemeingültige Definition zu finden ist, auch wenn sich der Grundgedanke meist sehr ähnelt, soll kurz vorgestellt werden, was im Rahmen dieser Arbeit unter einer Softwarearchitektur verstanden wird.

> „Eine Softwarearchitektur beschreibt die Struktur eines Softwaresystems. Diese umfasst die statischen Strukturelemente (z.B. Komponenten, Module), ihre extern sichtbaren Eigenschaften (z.B. Attribute, Methoden) sowie ihre nach außen sicht- und nutzbaren Schnittstellen und Interaktionen.“

Aus dieser Definition wird klar erkennbar, welche Elemente und Informationen für eine Bewertung der Softwarearchitektur überhaupt zur Verfügung stehen. Somit sind die Grenzen der objektiven Bewertung klar absteckbar. Gleichwohl bestand im Rahmen der Untersuchungen ebenfalls die Möglichkeit bestimmte Informationen für die Darstellung (UML-Profil) und Datenablage (formales Datenmodell) zusätzlich zu fordern und als dauerhafte Erweiterung der beiden genannten Dokumentationsformen fest zu etablieren.

3.1 Das Qualitätsmodell

Basis für die konkreten zu entwickelnden und zu implementierenden Metriken bildet zunächst eine Übersicht über die Gesamtheit aller grundsätzlich möglichen Qualitätskriterien. Darin sind zunächst sowohl funktionale als auch nicht-funktionale Kriterien enthalten, wobei im Folgenden der Fokus klar auf die nicht-funktionalen gelenkt werden soll. Bei der Sammlung der Kriterien konnte auf bestehende Ansätze zur Bewertung von Softwarequalität zurückgegriffen werden [Bo78][Lo03][Mc94][ISO01][Oe06][Ra93]. Ergebnis ist das in Abb. 2 dargestellte, so genannte *Qualitätsmodell*, welches sieben Oberkriterien mit jeweils mehreren (mit Ausnahme der Konformität) Unterkriterien enthält. Oberkriterien, von jetzt an auch als *Qualitätskriterien* bezeichnet, sind nicht direkt messbar sondern fassen vielmehr ein oder mehrere direkt messbare Kriterien, von jetzt an auch als *Qualitätsattribute* bezeichnet, zu einer Obergattung zusammen. Die obigen Ansätze schlagen die Kriterien wie erwähnt primär für den Einsatz zur Bewertung von Software vor, die Bewertung einer Softwarearchitektur kann in diesem Falle jedoch als Spezialfall einer Software-Bewertung aufgefasst werden, so dass die Kriterien grundsätzlich kompatibel sind.

Das hier vorgestellte Qualitätsmodell kombiniert, erweitert und ergänzt die aus den unterschiedlichen Ansätzen entnehmbaren Kriterien und spezialisiert diese im Anschluss daran an die speziellen Bedürfnisse des vorliegenden Anwendungsfalls. Dabei handelt es sich um Softwarearchitekturen von Automotive Embedded Software, so dass eine zweifache Spezialisierung vorgenommen wird. Dies ist für alle Kriterien und Attribute notwendig, weil diese je nach Anwendungsbereich (Domäne) stark unterschiedlichen Charakter aufweisen können und somit nicht direkt miteinander vergleichbar bzw. übertragbar sind. Ein Beispiel hierfür ist die Skalierbarkeit, welche in Softwareprodukten vieler Anwendungsbereiche ein wesentliches Qualitätsmerkmal darstellt, aber aufgrund der stark unterschiedlichen Eigenarten dieser Domänen ganz unterschiedlich ausgeprägt ist. Während beispielsweise in webbasierten Datenbankanwendungen Zugriffszeiten und Datendurchsätze sich entsprechend der Skalierung der Datenbank verhalten sollten, steht im Automobilbereich das ressourceneffiziente Hinzufügen und Entfernen von Funktionalität für unterschiedliche Kunden- oder Plattformausprägungen von bestimmten Funktionen/Systemen im Vordergrund, um je nach Ausprägung mit unterschiedlich leistungsstarken Plattformen (Steuergeräten) auszukommen und somit Kosten zu sparen. Die Skalierbarkeit äußert sich somit in der einfachen Entfern- und Hinzufügbarkeit von Funktionalität und baut somit auf einen modularen Aufbau von Software. Solche speziellen Randbedingungen gilt es für alle Kriterien zu berücksichtigen.

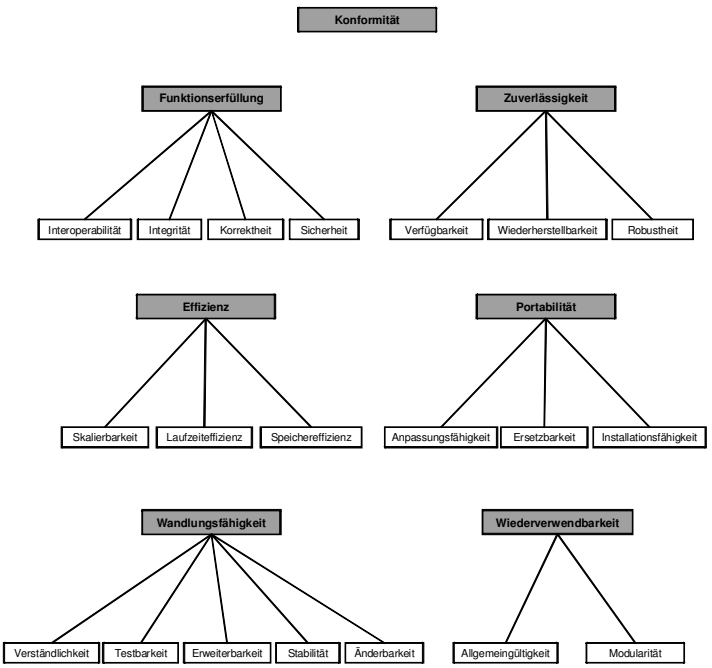


Abbildung 2: Qualitätsmodell zur Klassifizierung und Bewertung von Softwarearchitektur

Neben der Definition und Anpassung ist ein weiterer wesentlicher Schritt die Relevanz und Bedeutung der Kriterien und Attribute für die Automobildomäne zu ermitteln, um so eine Priorisierung in Hinblick auf die Entwicklung der Metriken vornehmen zu können.

Eine Einteilung der Qualitätskriterien kann entsprechend der drei Faktoren *Einsatzbedingung*, *Produktionskosten* und *Entwicklungskosten* vorgenommen werden (Abb. 3).

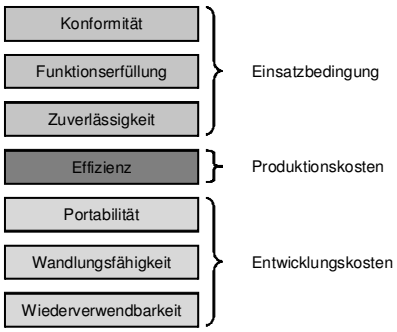


Abbildung 3: Relevante Qualitätskriterien für den Automotive-Bereich

In der BMW Group existieren bereits verschiedene Verfahren und Prozesse, um die drei Qualitätskriterien Konformität, Funktionserfüllung und Zuverlässigkeit sicherzustellen. In der vorliegenden Arbeit wird daher der Fokus zur Entwicklung von Metriken auf die zwei Schwerpunkte Produktions- und Entwicklungskosten gelegt. Eine objektive Bewertung unterschiedlicher Architekturentwürfe soll, wenn möglich, nach den Kriterien *Effizienz*, *Wandlungsfähigkeit* und *Wiederverwendbarkeit* vorgenommen werden.

3.2 Die objektiven Bewertungsmetriken

Ziel aller Architekturmetriken ist das messbare beziehungsweise formale „greifbar machen“ von vormals subjektiven Designentscheidungen, Architekturmustern und Best Practices. Es galt somit das subjektive und schwer zugängliche Wissen erfahrener Softwarearchitekten „anzuzapfen“ und in eine reproduzierbare, objektive und quantifizierte Form zu überführen. Dadurch wird eine Korrelation zwischen subjektiven und objektiven Kriterien und Entwurfsmustern geschaffen. Die Basis für die Metriken waren somit intensive Fachgespräche und Diskussionen mit entsprechenden Mitarbeitern des Unternehmens. Über standardisierte Fragebögen und das Durchgehen konkreter (für alle Interviews identischer) Fallbeispiele wurde die Grundlage zur Messbarkeit geschaffen.

Durch die Gespräche hat sich gezeigt, dass grundsätzlich für nahezu alle der relevanten Attribute der drei Kriterien eine Möglichkeit besteht mit Hilfe der in der Softwarearchitektur zur Verfügung stehenden Informationen eine quantitative Metrik herzuleiten. Lediglich für die Kriterien *Erweiterbarkeit* und *Allgemeingültigkeit* ließ sich zum jetzigen Zeitpunkt kein geeignetes Verfahren finden. Dies liegt in beiden Fällen daran, dass reines Strukturwissen nicht ausreichend ist, um eine zuverlässige Aussage zu treffen. Jede Architektur ist grundsätzlich erweiterbar, wie „gut“ oder einfach allerdings eine solche Erweiterung vorgenommen werden kann, liegt ganz konkret an der Funktionalität, die hinzugefügt werden soll. Somit ist Kontextinformation, mit anderen Worten Detailwissen über mögliche künftige Erweiterungen, zwingend erforderlich. Ähnlich verhält es sich mit der Allgemeingültigkeit, die ebenso nicht ausschließlich auf Strukturinformationen fußend bewertet werden kann.

Die im Rahmen dieser Arbeit entwickelten, objektiven Softwarearchitektur-Metriken finden sich in Abb. 4. Auf zwei ausgewählte Metriken, *Strukturverständlichkeit* sowie *Modularität*, soll im Folgenden kurz eingegangen werden, um einen Eindruck über Umfang und Charakter der Metriken zu vermitteln.

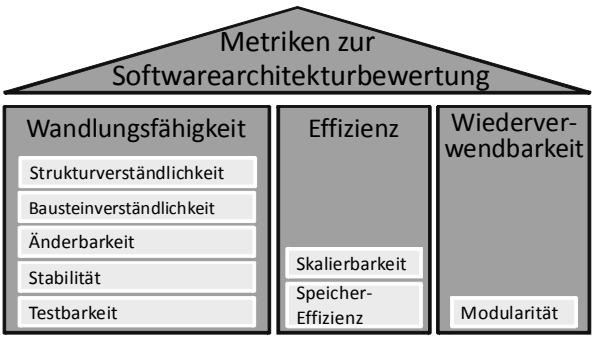


Abbildung 4: Übersicht über die entwickelten und implementierten Metriken des Qualitätsmodells

Metrik zur Strukturverständlichkeit:

Idee:
Die Struktur der Softwarearchitektur hat einen großen Einfluss auf ihre Verständlichkeit für die beteiligten Stakeholder beziehungsweise Entwickler. Durch Abstraktion und Verfeinerung können Softwarekomponenten in Subkomponenten zerlegt oder zu übergeordneten Bausteinen zusammengefasst werden. So entstehen verschiedene Sichten mit unterschiedlichem Detaillierungsgrad. Nach [Mi56] sind Systeme, die aus 7±2 Subsystemen bestehen, für einen Menschen optimal verständlich. Daher soll in dieser ersten Metrik unter der obigen Prämisse die Anzahl der Subsysteme über die gesamte Softwarearchitektur bewertet werden.

Metrik:
Um eine Gewichtung und Eingrenzung des Wertebereichs vornehmen zu können, wird für die Anzahl der Subsysteme einer Komponente eine Klassifizierung vorgenommen. Zu diesem Zweck wird der *Component Complexity Factor (CCF)* eingeführt, welcher sich nach Tabelle 1 zusammensetzt. Der CCF wiederum wird in der Berechnungsvorschrift der Metrik benötigt.

Tabelle 1: Berechnung des Component Complexity Factors

# Subkomponenten	CCF
0	0
1-4	2
5-9	0
10-11	3
12-14	5
>14	10

$$M1_i = \frac{CCF_i + \frac{\sum_{j=1}^{n} M1_j}{n}}{2}$$

$1 \leq j \leq n$
i = aktuelle Komponente
n = Anzahl aller Subkomponenten von i
CCF_i = Component Complexity Factor von Komponente i
$M1_i$ = Strukturverständlichkeit der aktuell untersuchten Komponente i
$M1_j$ = Strukturverständlichkeit von Subkomponente j

Erläuterungen:
Die Metrik arbeitet rekursiv über alle Hierarchieebenen hinweg. In der aktuellen Hierarchieebene wird ein Wert für die Strukturverständlichkeit des jeweiligen Strukturelements gebildet und zur Ebene darüber hochgegeben, um dort mit den Werten der übrigen Bausteine des gleichen Levels gemittelt zu einem Gesamtwert des umgebenden Strukturelements verarbeitet zu werden. Der Wertebereich dieser Metrik liegt zwischen 0 und 10. Aus Platzgründen kann an dieser Stelle leider nicht eingehender auf die Metrikanwendung eingegangen werden.

Metrik zur Modularität:

Idee:
Eine gute Modularität zeichnet sich durch eine starke Kohäsion und schwache Kopplung aus [BC99]. Logisch zusammenhängende Softwaremodule, welche viele Informationen austauschen, sollten demnach zu einer Komponente zusammengefasst werden. Eine starke Zusammengehörigkeit und schwache äußere Abhängigkeit kann in der modellbasierten Entwicklung vor allem über das Verhältnis interner zu externer Kommunikation überprüft werden. Dies gilt für alle Strukturelemente auf beliebigen Ebenen.

Metrik und Erläuterungen:
Für die Modularitäts-Metrik wird kein zusätzlicher Faktor benötigt, maßgeblich ist für jeden Baustein allein das Verhältnis zwischen Botschaften, die auf derselben Hierarchieebene ausgetauscht werden, zu der Gesamtzahl aller (internen und externen) Botschaften. Zunächst ermittelt die Formel das Verhältnis zwischen interner und externer Kommunikation einer Komponente. Mit einer Gewichtsverteilung von 50:50 fließen die Modularitätswerte der gegebenenfalls vorhandenen untergeordneten Komponenten – ebenso durch ein rekursives Verfahren – in die Berechnung ein. Der Wertebereich liegt zwischen 0 und 1.

$$M7_i = \frac{\frac{\sum Mess_{ext,i}}{\sum Mess_{ext,i} + \sum Mess_{int,i}} + \frac{\sum_{j=1}^{n} M7_j}{n}}{2}$$

$1 \leq j \leq n$

i = *aktuelle Komponente*

n = *Anzahl aller Subkomponenten von i*

$Mess_{ext,i}$ = *ein- oder ausgehende Schnittstelle der Komponente i zu einer anderen Komp.*

$Mess_{int,i}$ = *Schnittstelle zwischen Subkomponenten der Komponente i*

$M7_i$ = *Modularität der aktuell untersuchten Komponente i*

$M7_j$ = *Modularität von Subkomponente j*

3.3 Anwendung der Metriken

Alle Metriken können grundsätzlich unabhängig voneinander eingesetzt werden. Dies ermöglicht einen flexiblen und individuellen Einsatz je nach den Bedürfnissen des bewertenden Entwicklers. Für eine gesamthafte Architekturbewertung werden alle einzelnen Metriken berechnet und in einer bestimmten Gewichtung ins Verhältnis gesetzt. Auch diese Gewichtung ist im Tool (Kapitel 4) individuell einstellbar, als Ausgangsgewichtung wurde die in Tabelle 2 dargestellte Gewichtung vorgenommen, welche aus den subjektiven Einschätzungen der befragten Entwickler zur Wichtigkeit abgeleitet wurde.

Tabelle 2: Initiale Gewichtung aller Kriterien und Attribute

Kriterium	Gewicht	Metrik	Gewicht
Wandlungsfähigkeit	2	Strukturverständlichkeit	1
		Bausteinverständlichkeit	1
		Änderbarkeit	3
		Stabilität	1
		Testbarkeit	2
Effizienz	2	Skalierbarkeit	2
		Speicher-Effizienz	1
Wiederverwendbarkeit	1	Modularität	1

Auch wenn eine absolute Bewertung einer einzigen Architektur problemlos möglich ist, hat sich gezeigt, dass ein relativer Vergleich zweier Vergleichsarchitekturen zweckmäßiger ist. Dies liegt zum einen daran, dass die Metriken prinzipbedingt unterschiedliche Wertebereiche haben, was ein Gewichten zu einem einzelnen absoluten Gesamtwert erschwert. Zum anderen ist in der frühen Phase des Softwarearchitektur-Entwurfs, zu der oftmals noch gar nicht die endgültigen (funktionalen und nicht-funktionalen!) Anforderungen an das Softwaresystems vorliegen, eine isolierte Einordnung der quantitativen Metrikwerte schwierig. Erst mit steigender Erfahrung durch zahlreiche Architekturbewertungen kann für vergleichbare Projektumfänge auch über den einzelnen Zahlenwert eine Einschätzung der Architekturgüte vorgenommen werden. Die Autoren schlagen daher zunächst eine vergleichende Architekturbewertung vor, bei der grundsätzlich immer eine Referenz- und eine Vergleichsarchitektur bewertet werden. Die Metriken werden im Hintergrund für jede Architektur berechnet, anschließend wird eine prozentuale Veränderung angegeben. Dadurch wird die Problematik der unterschiedlichen Wertebereiche umgangen, es kann somit auch ein (gewichteter) Gesamtwert der relativen Verbesserung ermittelt werden.

In den bisherigen Bewertungen von Softwarearchitekturen hat sich dieses Verfahren bewährt und dient zukünftig der Feinabstimmung der Metriken in Hinblick auf Wertebereich, Faktoren und Gewichtung.

4 Softwareprototyp und Praxisbeispiel

Alle acht vorgestellten (Attribut-)Metriken sind inzwischen in einem Softwareprototyp umgesetzt. Das Werkzeug dient momentan in der Entwicklung von Fahrerassistenzsystemen der Sammlung von Erfahrungen im Umgang mit objektiven Softwarearchitekturbewertungen und dient der Evaluierung des Ansatzes. Aufgrund der automatischen Überführung der Softwarearchitektur in das angesprochene Datenformat können auch große und komplexe Softwarearchitekturen von realen Fahrerassistenzsystemen schnell und komfortabel bewertet werden.

Der Softwareprototyp wurde mit der Open-Source Entwicklungsumgebung Eclipse auf Java-Basis umgesetzt und bedient sich der JAXB-Technologie [SUN03]. JAXB ist eine Programmierschnittstelle in Java, welche es ermöglicht XML-Dateien, die einem XML-Schema konform aufgebaut sind, durch einen so genannten „unmarshall-Vorgang“ in Java-Klassen zu überführen. Durch herkömmliche Java-Programmierkonstrukte werden die in den Klassen hinterlegten Struktur- und Attributinformationen ausgelesen und entsprechend der vorgegebenen Metrik-Algorithmen verarbeitet. Die Bedienung erfolgt für den Benutzer komfortabel mit einer grafischen Bedienoberfläche (GUI). Dort können die zu bewertenden XML-Artefakte (Softwarearchitekturen) eingestellt und die Bewertung individuell konfiguriert werden. Die zu berechnenden Einzelmetriken sowie deren Gewichtung können ebenso frei eingestellt werden. Das Tool wurde in Form einer jar-Datei so umgesetzt, dass es auf beliebigen (Java-)Plattformen eigenständig lauffähig ist.

Eine beispielhafte und tabellarisch aufbereitete Bewertung von zwei Softwarearchitekturvarianten ist im Folgenden kompakt dargestellt, die Ergebnisse sind als relative Änderung aufgelistet (Tabelle 3). Referenzarchitektur ist die aktuelle Softwarearchitektur der Fahrerassistenzsysteme der Längsdynamik der BMW Group, Vergleichsarchitektur ist ein erster Entwurf zur Neustrukturierung und Optimierung. In diesem konkreten Fall wurden gezielt wenige Maßnahmen durchgeführt, welche sich nur auf die beiden vorgestellten Teilmetriken auswählen. Dabei handelt es sich um strukturelle Maßnahmen zur Änderung der Komposition von feingranularen zu grobgranularen Bausteinen, welche sich sowohl auf die Strukturverständlichkeit (‚Aufbau und Anzahl von enthaltenen Subkomponenten‘) als auch die Modularität (‚innere zu äußere Kommunikation‘) auswirken. Es bietet sich augenscheinlich zum Kennenlernen der Metriken – z.B. Ausnutzung des Wertebereichs, Sensitivität etc. – an, kleinere, isolierte Umstrukturierungsmaßnahmen vorzunehmen und einzeln zu bewerten.

Dies entspricht dem grundsätzlich iterativ-inkrementellen Prozess des schrittweisen Softwarearchitektur-Entwurfs, bei welchem die vorgestellte objektive Softwarearchitekturbewertung künftig einen entscheidenden Stellenwert einnehmen soll. Kleinere Werte entsprechen dabei grundsätzlich Verbesserungen, somit sind im relativen Vergleich zweier Architekturvarianten negative Vergleichswerte anzustreben.

Tabelle 3: Exemplarische relative Bewertung von zwei Softwarearchitekturvarianten

Metrik		Werte-bereich	Gewicht	Referenz-SWA	Umstrukturierung	
				Wert	Wert	Diff.
Verständlichkeit	M1	0-10	1	0,645	0,570	-11,63%
Verständlichkeit	M2	0-60	1	11,370	11,370	0,00%
Änderbarkeit	M3	0-240				
Stabilität	M4	0-1	1	0,588	0,588	0,00%
Testbarkeit	M5	0-240	2	23,926	23,926	0,00%
Skalierbarkeit	M6	0-50		17,481	17,481	0,00%
Modularität	M7	0-1		0,470	0,424	-9,79%

Qualitätskriterium	Gewicht	Wert
Wandlungsfähigkeit	2	-2,33%
Effizienz	2	0,00%
Wiederverwendbarkeit	1	-9,79%
		-2,89%

5 Zusammenfassung und Ausblick

Der in dieser Arbeit vorgestellte Ansatz beschäftigt sich mit den Möglichkeiten zur objektivierten Bewertung von Automotive Embedded Softwarearchitekturen am konkreten Beispiel der Fahrerassistenz. Es wird eine Methode vorgestellt, welche in dieser Form erstmalig Metriken bereitstellt, mit denen derartige Softwarearchitekturen *objektiv* und *quantifiziert* bewertet werden können. Zu diesem Zweck wurde ein *Qualitätsmodell* entwickelt, welches aus bekannten Ansätzen zur Softwarebewertung Kriterien und Attribute zusammenträgt und für den speziellen Anwendungsfall spezialisiert und anpasst. Für die Mehrzahl der für den Automotive-Bereich relevanten Attribute wurden insgesamt acht Metriken entwickelt und an ausgewählten Beispielen vorgestellt.

Die Metriken ermöglichen eine automatisierte Bewertung einer in einem vorgegebenen selbst entwickelten Datenformat vorliegenden Softwarearchitektur. Dabei sind sowohl absolute als auch relative Architekturbewertungen möglich. Die Metriken wurden in einem Softwareprototyp bereits umgesetzt und können somit produktiv für die Entwicklung von iterativ und zielgerichtet optimierten Softwarearchitekturen der Fahrerassistenz eingesetzt werden. Verschiedene Anwendungsbeispiele runden den Beitrag ab.

Die Überführung subjektiver Entwurfsmuster und Intuitionen in objektivierte Metriken gestaltet sich erwartungsgemäß schwierig. Die entwickelten Metriken stellen somit eine Ausgangsbasis für weitere Untersuchungen dar. Erst durch zunehmende Erfahrung im Umgang mit dem Bewertungsverfahren kann die Korrelation zwischen subjektiver und objektiver Bewertung verbessert werden. Die Erfahrungen sollen helfen, die Faktoren und Gewichte der Metriken besser abzustimmen. Dadurch wird erreicht, dass sich bei vorgenommenen Änderungen der Architektur die Werte der einzelnen Metriken angemessen und konsistent zueinander ändern. Erst dann erreichen die Metriken eine Reife, die auch absolute Bewertungen einzelner Architekturentwürfe ermöglicht. Für jedes einzelne Attribut werden so Erfahrungswerte je nach Domäne und Komplexität der Softwarearchitektur ermittelt, die eine Einordnung von absoluten Zahlenwerten erst ermöglichen.

In Zukunft sollen die Metriken verstärkt in den Software-Entwicklungsprozess eingebunden werden, um eine effektive und effiziente Entwicklung von Automotive Softwarearchitekturen zu unterstützen. Die dadurch ermöglichte reproduzierbare und objektive Untersuchung der Architekturen auf ihre Eignung und Güte unterstützt die Entwickler beim Erreichen der Vielzahl nicht-funktionaler Anforderungen wesentlich.

Literaturverzeichnis

[Ab96] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Zaremski: "Recommended Best Industrial Practice for Software Architecture Evaluation," Georgia Institute of Technology, Carnegie Mellon University, Software Engineering Institute, 1996

[Ah08] D. Ahrens, A. Frey, A. Pfeiffer, T. Bertram: „Entwicklungsprozess und abstrakte Softwarearchitektur für modulare Automotive Software," 3. Dortmunder Autotag, 2008

[Ah09] D. Ahrens, A. Frey, A. Pfeiffer, T. Bertram: „Entwicklung einer leistungsfähigen Darstellung für komplexe Funktions- und Softwarearchitekturen im Bereich Fahrerassistenz," VDI Mechatronik 2009, Wiesloch, 2009

[Ah10] D. Ahrens, A. Frey, A. Pfeiffer, T. Bertram: „Designing Reusable and Scalable Software Architectures for Automotive Embedded Systems in Driver Assistance," to appear at SAE World Congress, Detroit, USA, 2010

[APB08] D. Ahrens, A. Pfeiffer, T. Bertram: „Comparison of ASCET and UML - Preparations for an Abstract Software Architecture," Forum on Specification and Design Languages (FDL) 2008, Proceedings, Stuttgart, Germany, 2008

[BC99] C. Baldwin, K. Clark: „Design Rules: The Power of Modularity Volume 1," Cambridge, MA, USA, MIT Press, 1999

[Bo78] B. Boehm, J. Brown, H. Kaspar, M. Lipow, G. MacLeod, M. Merrit: „Characteristics of Software Quality," Elsevier Science Ltd, 1978

[ED96] C. Ebert, R. Dumke: „Software-Metriken in der Praxis," Springer Verlag, 1996

[EHM08] S. Eicker, C. Hegmanns, S. Malich: „Projektbezogene Auswahl von Bewertungsmethoden für Softwarearchitekturen," Multikonferenz Wirtschaftsinformatik, 2008

[ISO01] ISO - International Standards Organization: „Software engineering — Product quality," ISO 9126-1, 2001

[Lo03] F. Losavio, L. Chirinos, N. Levy, A. Ramdane-Cherif: „Quality Characteristics for Software Architecture," Journal of Object Technology Vol. 2, No. 2, 2003, S. 133-150

[Mc94] J. McCall: „Encyclopedia of Software Engineering," 1994, S. 958-969

[Mi56] G. Miller: „The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," The Psychological Review 63, 1956, S. 81-97

[Oe06] K. Oey: „Nutzen und Kosten von serviceorientierten Architekturen," Universität Köln, Dissertation, 2006

[OMG05] OMG – Object Management Group: UML – Unified Modeling Language, www.uml.org

[PBG07] T. Posch, K. Birken, M. Gerdom: „Basiswissen Softwarearchitektur," dpunkt.verlag GmbH, 2007

[Ra93] J. Raasch: „Systementwicklung mit strukturierten Methoden. Ein Leitfaden für Praxis und Studium," Hanser Verlag, 1993

[RH06] R. Reussner, W. Hasselbring: „Handbuch der Software-Architektur," dpunkt.verlag GmbH, 2006

[SUN03] SUN: JAXB - Java Architecture for XML Binding, https://jaxb.dev.java.net

[W3C98] W3C – World Wide Web Consortium: XML – Extensible Markup Language, www.xml.org

Multi-Level Test Models for Embedded Systems

Abel Marrero Pérez, Stefan Kaiser
Daimler Center for Automotive IT Innovations
Technische Universität Berlin
Ernst-Reuter-Platz 7, 10587 Berlin
abel.marrero@dcaiti.com, stefan.kaiser@dcaiti.com

Abstract: Test methodologies for large embedded systems fail to reflect the test process as a whole. Instead, the test process is divided into independent test levels featuring differences like the functional abstraction levels, but also similarities such as many functional test cases. Desirable instruments such as test front-loading feature a considerable test effort and test cost reduction potential, but their efficiency suffers nowadays from the strict separation of the test levels and the consequent lack of appropriate mechanisms for reusing tests across test levels. Multi-level test cases have shown to provide the means for a seamless test level integration based on test case reuse across test levels. This paper extends this concept by introducing multi-level test models which are capable of systematically integrating different functional abstraction levels. From these models, we can derive multi-level test cases that are executable at different test levels. With this novel approach, multi-level testing benefits from the principles of model-based testing while the requirements for providing multi-level capabilities to any test models are analyzed and described.

1 Introduction

The complexity of large embedded systems leads to particularly long development and testing cycles. The long time span between the availability of the first software components and the first system prototype (e.g. an automobile) highlights the importance of testing earlier on in the development process. In this context, the introduction of test front-loading is essential, since the sooner a fault is revealed, the cheaper its correction becomes.

Test front-loading addresses the earlier execution of test cases conceived for higher test levels. For instance, a system test case might be executed on a software component. Common lifecycle models such as the V model do not explicitly consider front-loading. Moreover, their strict separation of test levels constitutes an obstacle for systematic and efficient test front-loading.

We thus call for ending this strict separation by integrating test levels. With test level integration we do not aim at merging test levels but at bringing them closer together. The idea is to establish relations between test levels that support a systematic test front-loading. For this purpose, we need a new test methodology that takes the whole V model into account rather than single test levels. An additional objective is to reduce the effort necessary for test specification, design, implementation and maintenance.

Effort reduction for these test activities has become a major issue after the notable advances in test automation of the last decade. Much of the optimization potential of test automation has now been tapped. In contrast, new methods focusing on optimizing the mentioned testing activities promise significant efficiency gains. We focus on these activities and pay particular attention to test level integration. For testing embedded systems, we expect to additionally benefit from new approaches in this field due to the fact that more test levels exist for testing embedded systems than for testing standard software.

We have proposed multi-level test cases as an instrument for supporting functional test case reuse across test levels as well as test level integration (cf. [MPK09a]). Such test cases consist of two parts. Firstly, a test level-independent test case core that is reused across test levels. Secondly, test level-specific test adapters. The structure of multi-level test cases supports the separation of commonality and variability across test levels.

In this paper, we introduce multi-level test models as an extension of this concept. With this approach we take advantage of the numerous synergies within a test suite in order to create abstract test models from which concrete multi-level test cases can be derived. With this novel approach we aim at benefiting from model-based techniques as well as further reducing the test effort, particularly with regard to the aforementioned test activities.

The headlights example presented in the next section will serve as an example throughout this paper. Section 3 details the reasons for applying reuse techniques across test levels. A brief overview of multi-level test cases follows before multi-level test models are introduced in Section 5. We conclude the paper with a related work overview and a summary.

2 Example

Our running example concerns the headlights function of a modern vehicle. As shown in Fig. 1, four different electronic control units are involved: the driver panel (DP), the ignition switch control (ISC), the light sensor control (LSC) and the automated light control (ALC). The ALC is the master component. It decides whether to turn on the headlights depending on the light switch position, the ignition switch position and the outside illumination. This information is received via a CAN bus.

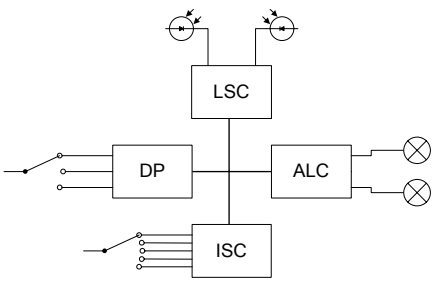


Figure 1: Electronic Control Units Involved in the Headlights Function

3 Test Reuse Across Test Levels

In the introduction we mentioned that the test process for embedded systems features additional test levels. We will consider four different test levels in this paper: software component testing, software integration testing, software/hardware integration testing, and system integration testing (cf. Fig. 2). In terms of functionality, software component testing is the most detailed test level, while system integration testing represents the most abstract one. These test levels reflect a large integration process starting with software components and ending with the complete system consisting of a set of electronic control units. We have described these test levels in-depth in [MPK09a].

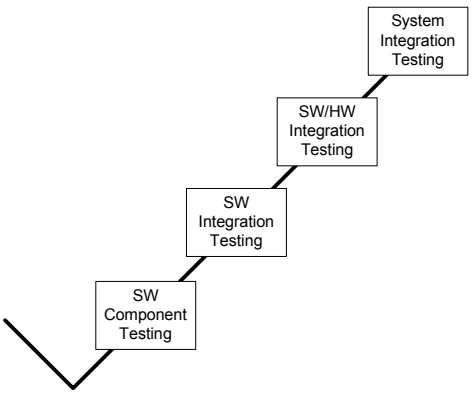


Figure 2: Right Branch of the V Model for Embedded Systems

The V model [SL07] treats each test level independently. The model recommends considering the development documents originated at the corresponding development level on the V model's left branch for creating test cases. This is a good practice since the functional abstraction levels of both development and test levels are the same. The V model does not consider test front-loading and further similarities between the test levels, however. We need test front-loading for an early assurance of the development artifacts' quality.

Our running example provides evidence of the significance of front-loading. A basic test case for the vehicle's headlights might consist of two steps: turning the headlights on and off using the light switch. During development, this test case can be executed once the first vehicle prototype has been built using the light switch and visually observing whether the headlights react. The prototype will be built relatively late in development, so that finding a fault in the prototype that could have been found earlier would be a costly exercise.

The idea of front-loading is to start executing this test case as soon as possible. We could take the electronic control unit driving the actual lamps and check if the headlights can be turned on and off. We might not have a switch available at this test level if the switch is wired to another control unit. Hence, we might have to stimulate a CAN bus at software/hardware integration test level. We can port the test case further down towards the software. We will find a software component including the basic logic that decides whether

to turn on the headlights depending on the switch position. In fact, if the software is developed model-based, we can even test these functional models, as well. With this practice, which intrinsically considers test case reuse across test levels, we will be assuring at every test level that the headlights will later work based on the components integrated so far.

Systematically applied front-loading leads to revealing only those faults at every test level that are directly related to the integration step performed by that test level. If the headlights do not turn on at software/hardware integration test level, but did at software integration test level, there might be a problem with the hardware or its interface to the software.

A test case might specify that the switch has to be turned on, but at most test levels no switch will be available. The test interface, i.e. the concrete information channels that are used for both stimulating and observing the test object, will typically be different for each test level, because at each test level there are different test objects. Consequently, only test cases that abstract from the test interface will really be reusable. This is the case, for instance, for informal textual test specifications. In other words, we can reuse the headlights test case presented above as long as we do not concretely specify which signals with which coding are used for test stimulation or observation.

Handling different interfaces is inevitable for test implementations, however. Hence, a dedicated approach for reusing test cases is necessary for practicing front-loading systematically and with low effort. Such an approach has to take the different test objects and test interfaces into consideration. Another fundamental difference between test levels is the functional abstraction level of the test objects. Software components are very concrete because they include many implementation details. On the other hand, system integration testing represents the most abstract test level.

Acquiring the capability of dealing with these reuse obstacles will report several advantages in terms of test effort reduction. Test reuse across test levels is essential for an efficient front-loading, but it also has the potential to reduce the test suites' size at the different test levels. Similar or even identical test cases across test levels could be reused and thus only implemented and maintained once.

In summary, we can state that it is desirable to apply front-loading in order to find faults earlier. This implies reusing test cases across test levels which is directly possible if the test interface is omitted (e.g. for textual test specifications). For test design and implementation there is a set of barriers mainly regarding the test interface that needs to be sorted out before these artifacts can be reused. We propose using multi-level test cases for this purpose.

4 Multi-Level Test Cases

We introduced multi-level test cases in [MPK09a] as an instrument for test level integration. Reusing a test case implementation across up to four different test levels instead of implementing separate test cases for each test level promises great effort reductions.

Front-loading requires additional test effort in order to find faults earlier. It is necessary to reduce this additional effort as much as possible in order to benefit from this practice.

Completely reusing test cases is impossible due to the differences in functional abstraction and test interfaces across test levels. Hence, the goal is to achieve the highest possible reuse percentage. For this purpose, we distinguish two different modules within test cases: a reusable test case core (TCC) containing the actual test behavior and a variable test adapter (TA) specifically designed for a concrete test level or even test object. Fig. 3 shows the structure of multi-level test cases.

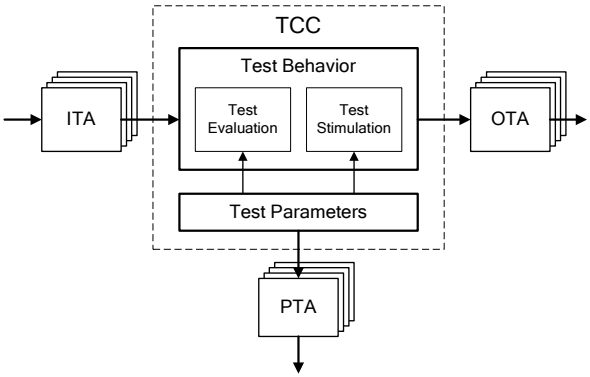


Figure 3: Structure of Multi-Level Test Cases

The test behavior in the TCC is invariant across test levels, i.e. test level-independent. It is abstract and can be reused across test levels. It possesses a stimulation interface and an observation interface for interacting with the system under test. The only form of variability accepted within the TCC are test parameters, which are necessary for bridging differences between test levels such as the level of accuracy (cf. software vs. hardware) or the kind of execution (e.g. real-time vs. simulation). These differences only imply minor variations within the test behavior that do not affect its intention.

In contrast, the TA varies across test levels and is thus test level-specific. It is responsible for bridging the (major) differences between the TCC and the test object at a certain test level. The test adapter consists of three parts: the input test adapter (ITA), the output test adapter (OTA) and the parameter test adapter (PTA).

Both ITA and OTA adapt the TCC interface to the test object interface. They thus map the abstract test behavior to the test object at its abstraction level. In the headlights example, reusing a system integration test case stimulating the light switch at software/hardware integration test level will require mapping the switch stimulation to an equivalent stimulation of the CAN bus. The OTA thus has to emulate the test case-related functionality of any components located between the switch and the CAN bus in the real system.

Both ITA and OTA basically have to model the behavior of components that are not available at a certain test level but whose presence is expected by the TCC. The behavior of such missing components does not necessarily have to be modeled completely – a partial behavior model covering the needed interface signals is sufficient (test case-related behavior). The PTA has to map any test parameters that apply to the test object.

Multi-level test cases can also be applied for reusing test cases from lower test levels at more abstract test levels. In this case, both ITA and OTA will have to model the inverse behavior of all components that are added along the integration process. In this context, more advanced mechanisms will have to be applied to maintain causality in case these additional components introduce delays. Delay balancing and delay compensation constitute a solution for this problem [MPK09b].

We can summarize that the key idea of multi-level test cases is to separate the test behavior into an abstract part within the TCC and an abstraction (ITA) or refinement (OTA) of this abstract behavior to a concrete test level featuring different test objects with different interfaces. This separation makes the reuse of major parts of the test cases possible, which is an essential requirement for efficient front-loading.

5 Multi-Level Test Models

Our model-based approach for designing multi-level test cases aims at creating a single multi-level test model for a concrete system function from which all multi-level test cases regarding that function can be derived. For example, instead of creating several multi-level test cases for testing the headlights, we create a multi-level test model and derive all multi-level test cases from it.

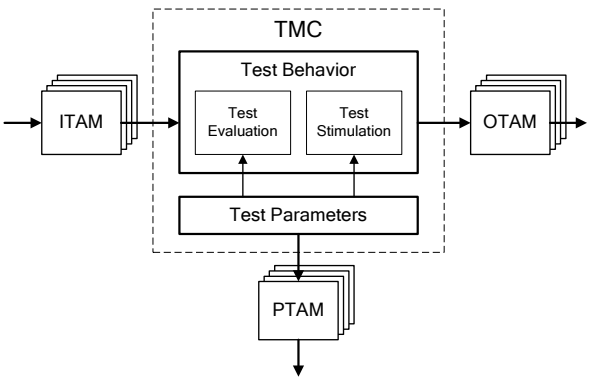


Figure 4: Structure of Multi-Level Test Models

The desired derivation of multi-level test cases from multi-level test models suggests maintaining the structure of the test cases for the test models. We thus propose separating the test model core (TMC) from the test adapter models (TAM) – as shown in Fig. 4. The TCCs will be derived from the TMC and the test adapters from the TAM.

This separation provides the same advantages as for multi-level test cases. For a concrete system function, only one test level-independent TMC has to be created. This TMC describes the test behavior and can be reused across all test levels. On the other hand, test level-specific TAMs have to be developed, i.e. single TAMs for each test level.

5.1 Test Model Core

While the TCC of multi-level test cases describes *partial* test behavior, the TMC will feature a more general behavioral description consisting of at least the composition of the test behavior of all test cases included in the textual functional test specification. Ideally, the TMC should describe the *complete* test behavior for a certain system function.

Note that test cases for a concrete system function will exist at different test levels. These test cases will belong to different functional abstraction levels. For instance, at the software/hardware integration test level there might be a test case stimulating a timeout of the CAN bus message containing information about the switch position. Furthermore, we assume that after $1\,s$ of not receiving this cyclic message, the CAN bus driver reports a timeout, and for safety reasons the headlights have to be turned on as long as the timeout persists. This test case is out of the system integration test level's scope because the handling of missing messages is too concrete for that test level. However, this test case can be reused at lower test levels (front-loading) in order to assure that the software reacts correctly to such a timeout before the first hardware prototype is available.

In consequence, the TMC will have to integrate test behavior at different functional abstraction levels. The alternative to this integration is to create a separate multi-level test model for each abstraction level. We opt for the integration approach in order to take advantage of the synergies between test cases across test levels. These synergies basically regard the test interface.

The TCCs of multi-level test cases created at different test levels often share part of the test interface. Some signals within the test interface are test level-specific. The switch message timeout represents an example of a test level-specific signal. Other signals are shared with higher abstraction levels, however. For example, the timeout test case mentioned above also observes the headlights, analogously to the basic headlights test case discussed in the previous sections consisting in turning the headlights on and off.

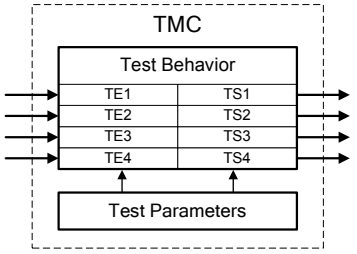


Figure 5: Division of the Test Behavior into Regions

We propose dividing the TMC into different regions – one for each abstraction level (cf. Fig. 5). Each region will be responsible for part of the TMC interface. Signals shared by different test levels will be assigned to the region corresponding to the most abstract test level. With this practice, the test behavior concerning a concrete signal is bundled in a single model region and does not have to be redundantly modeled in different regions.

For the derivation of a TCC at a certain abstraction level from a TMC, the region corresponding to the TCC abstraction level will be the most relevant. Furthermore, any more abstract regions might also be considered for *inheriting* additional signals. In contrast, less abstract regions will definitely be out of the derivation's scope. For instance, the basic on/off headlights test case introduced in Section 3 will be derived from the first model region only, since it belongs to the top test level. The timeout test case, however, will be derived from the first and second model regions. While the most relevant region will be the second, the headlights evaluation will occur in the first region, for example.

In some cases signals from different regions might need to be synchronized within test models. Appropriate mechanisms for performing this synchronization across regions have to be provided. By *appropriate* we mean effortless to a large extent, since this synchronization could otherwise constitute a disadvantage that counters the advantages of integrating the different abstraction levels of test behavior within a single test model.

5.2 Test Adapter Models

A test adapter model is a generalization of every possible test adapter for a concrete test object at a certain test level. Test adapter models are thus test behavior-independent. They only rely on the signals of the TMC interface and the TMC test parameters.

Test adapter models consist of three parts (cf. Fig. 4): an input test adapter model (ITAM), an output test adapter model (OTAM), and a parameter test adapter model (PTAM).

Our aim is to automatically generate the appropriate test adapter that maps a given TCC interface to a given test object interface from the corresponding test adapter model. This generation will be based on the interface of the system under test and the test object parameters. The generated test adapter will provide both the test object and the TMC with the necessary input signals and the test object with the appropriate parameter values.

The simplest ITAM will consist of a set of functions describing the input signals of the TMC as a function of the test object outputs. Analogously, a simple OTAM model will describe how to stimulate the test object inputs using the TMC outputs. These functions model any components lying between the interfaces being mapped.

An OTAM for the headlights example for the software/hardware integration test level will include a function mapping the switch position to a CAN bus message. This function partially models the behavior of the control unit attached to the switch. Additionally, it might contain other mappings such as that of the key position to the corresponding CAN bus message. For the timeout test case described above, we might need to map the switch position (for the beginning of the test case, before the timeout) but we might not need the key position, which might be needed for other test cases derived from the TMC. This will be taken into consideration for the test adapter generation. The generated test adapter will thus include the switch position mapping but exclude the key position mapping.

Fig. 6 shows an implementation example of an OTAM that can map any TCC at system integration test level to the interface of the ALC electronic control unit (which is at soft-

ware/hardware integration test level). It is a MATLAB/Simulink [MLS] model consisting of two lookup tables that map the switch signals to the CAN bus coding and a moving average filter that models the behavior of the LSC with respect to the outside illumination. The LSC provides an illumination signal without short peaks by filtering the original sensor data. The OTAM thus has to model this filtering in order to provide the CAN bus signal. Note that single test adapters do not necessarily have to include a filter. If a concrete TCC stimulates the light sensor with a constant value, the multi-level test case does not require the OTA to apply a filter since the filter output will equal its input. In analogy, the complete lookup tables will typically be unnecessary for a single test case. These examples demonstrate that test adapter models are more robust against changes than test adapters.

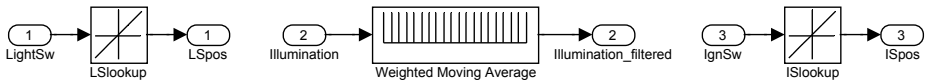


Figure 6: Output Test Adapter Model for SW/HW Integration Testing

Different test adapters can be generated from this OTAM. They will only differ in the signals used. For each test object input, the same test adapter will be used for any TCC because the OTAM is test behavior-independent, as we mentioned above. TAMs are hence similar to a mapping library containing different mappings ready for use.

Any test adapter generated from a test adapter model will correspond to a concrete test object at a certain test level. Mappings across multiple test levels can be achieved using more complex test adapter models but also concatenating multiple test adapter models mapping the interfaces of test objects at consecutive test levels. For example, once the switch position is mapped to the corresponding CAN bus message, we can reuse this mapping at software integration test level by mapping only the CAN bus message to the software variable representing the switch position. Test adapter model concatenation is particularly efficient for non-trivial mappings that require notable implementation effort as well as for frequently changing requirements and mappings (evolution). In those cases the concatenation provides a significant effort reduction (cf. test adapter concatenation in [MPK09a]).

5.3 Discussion

In this section, we have defined multi-level test models in a generic manner analogous to the multi-level test case definition. While multi-level test cases describe partial test behavior, multi-level test models cover the complete test behavior. The use of test level-specific TAMs leads to automatically adapting the TMC to the different test objects and abstraction levels along the test process. With this practice, the TMC can be reused across test levels without alteration – only test parameters might be subject to change.

Almost any test model can represent the TMC of a multi-level test model. We have found only four limitations to be mandatory in order to acquire the multi-level capability.

Firstly, multi-level test cases have to be derivable from multi-level test models. Secondly, updates within multi-level test models have to be automatically applied to the derived multi-level test cases. Thirdly, the TMC possesses a test interface consisting of a set of observation signals for test evaluation as well as a set of stimulation signals. Fourthly, the TMC has to cover different functional abstraction levels. The test interface is divided into different signal groups which are assigned to different regions within the test model. Each region corresponds to a certain test level.

This approach is deliberately kept generic, so that almost any test modeling approach can thus be employed for designing multi-level test cases. Dedicated tool support is only needed for the automatic generation of concrete test adapters from TAMs. A further criterion for the selection of a testing technology might be the support of test case portability across different test platforms at different test levels.

The presented approach takes advantage of all the benefits of model-based approaches such as better readability, reduced maintainability effort and a potentially increased functional coverage. Readability is crucial for front-loading and test reuse, since the resulting multi-level test cases will be shared by different parties at multiple test levels. Minimal maintainability effort is also of particular significance for large embedded systems due to the long development cycles which imply numerous changes. The functional coverage of multi-level test cases can increase by systematically modeling the overall test behavior.

On the one hand, a higher initial effort is typically required for creating multi-level test models in comparison to multi-level test cases – even in case there are large portions of commonality and synergies across test levels. On the other hand, once the models are created we can automatically generate concrete test adapters for any TCCs. Depending on the technology used, TCCs will also be automatically generated from the TMC.

The effort inversion in multi-level test models best pays off for large development cycles in which test cases and test objects are subject to continuous changes. The integration of different abstraction levels within the TMC is especially beneficial in terms of maintainability. For example, if the behavior of the headlights TMC input changes, it is sufficient to change one region of a single model in order to keep all multi-level test cases derived from that multi-level test model up-to-date.

6 Related Work

Front-loading has been proposed by Wiese et al. for early testing of system behavior in a model-based development context in the automotive domain [WHR08]. They suggest taking advantage of the early available functional models for testing system functionality. They also presented an approach for specifying test platform-independent test cases based on signal abstraction and a mapping layer. The use of abstraction for test case reuse is also proposed by Burmester and Lamberg [BL08]. Both approaches only consider reuse within a concrete abstraction level. However, they contribute notable ideas to our approach.

Beyond the front-loading context, Mäki-Asiala has denominated *vertical reuse* as the reuse of test cases across test levels [MA05]. He takes a pure reuse approach using TTCN-3

[TTC], i.e. he does not consider abstraction or refinement. He states that the similarities between test levels are fundamental for vertical reuse. We find that it is more important to assure that the differences are not substantial than completely focusing on the similarities.

In analogy to Schätz and Pfaller, Mäki-Asiala takes a bottom-up reuse approach and thus describes *interface visibility* as a central problem. Schätz and Pfaller solve this problem by automatically extending component test cases in order to gain system test cases that test the component at system test level [SP]. This test case extension is comparable to our test adapters. Schätz and Pfaller demonstrate that such test adapters can be automatically generated from formal component specifications. We cannot count on such accurate component specifications, especially for hardware components. Our approach therefore consists in approximating their behavior within test adapter models that focus on the complete test behavior instead of modeling the complete component behavior.

Our notion of test adapters is mainly based on the *adapter* concept introduced by Yellin and Strom in the context of component-based design [YS97]. The *platform adapter* and *system adapter* concepts within TTCN-3 are similar concepts in the testing domain. Lehmann (Bringmann) denominates *test engines* similar constructs within Time Partition Testing [LB03]. The main difference in comparison to our approach is that our test adapters are conceived for bridging abstraction differences across test levels.

The model-based approach for multi-level testing represents the major novelty within this paper. Model-based testing (MBT) nowadays constitutes a well-established research field and is gradually entering industry [Sch08], as well. Zander-Nowicka (Zander) provides an overview of MBT approaches paying particular attention to the automotive domain [ZNZ08]. Our approach is similar to the test models described in [PP05] or in [LB03]. In both approaches, test cases constitute partial behavior and are derived from abstract test models in the form of traces or variant combinations.

While reuse qualities are often attributed to model-based approaches [EFW02], to the best of the authors' knowledge no approaches have described a methodology for applying model-based testing for testing across test levels. The closest approach to ours in this context is TPT [LB03]. TPT features portability across test platforms [BK08].

7 Conclusion

This paper presents a novel model-based approach for systematically and efficiently testing large embedded systems at different test levels. Multi-level test models represent the model-based counterpart of multi-level test cases. We summarized the advantages of multi-level test models in Section 5.3. Our aim in this paper was to define a set of requirements that test models have to meet in order to become multi-level test models – independent of the modeling approach used.

We are currently applying multi-level test models to selected projects at Daimler AG using TPT. We plan on continuing this evaluation and reporting the results. In our future research we will focus on the automatic generation of test adapter models. We will also study efficient front-loading strategies that consider partitioning.

References

[BK08] Eckard Bringmann and Andreas Krämer. Model-Based Testing of Automotive Systems. In *1st International Conference on Software Testing, Verification, and Validation (ICST 2008)*, Lillehammer, Norway, 2008.

[BL08] Sven Burmester and Klaus Lamberg. Aktuelle Trends beim automatisierten Steuergerätetest. In *Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik II*, pages 102–111, Berlin, Germany, 2008. expert.

[EFW02] Ibrahim K. El-Far and James A. Whittaker. Model-Based Software Testing. In John J. Marciniak, editor, *Encyclopedia of software engineering*. Wiley, New York, NY, USA, 2002.

[LB03] Eckard Lehmann (Bringmann). *Time Partition Testing*. PhD thesis, Technische Universität Berlin, 2003.

[MA05] Pekka Mäki-Asiala. *Reuse of TTCN-3 Code*, volume 557 of *VTT Publications*. VTT, Espoo, Finland, 2005.

[MLS] The MathWorks, Inc. - MATLAB/Simulink/Stateflow. http://www.mathworks.com/.

[MPK09a] Abel Marrero Pérez and Stefan Kaiser. Integrating Test Levels for Embedded Systems. In *Testing: Academic & Industrial Conference - Practice and Research Techniques (TAIC PART 2009)*, pages 184–193, Windsor, UK, 2009.

[MPK09b] Abel Marrero Pérez and Stefan Kaiser. Reusing Component Test Cases for Integration Testing of Retarding Embedded System Components. In *First International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)*, pages 1–6, Porto, Portugal, 2009.

[PP05] Alexander Pretschner and Jan Philipps. Methodological Issues in Model-Based Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 281–291. Springer, Berlin, Heidelberg, Germany, 2005.

[Sch08] Hermann Schmid. Hardware-in-the-Loop Technologie: Quo Vadis? In *Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik II*, pages 195–202, Berlin, Germany, 2008. expert.

[SL07] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest*. dpunkt, Heidelberg, Germany, 3rd edition, 2007.

[SP] Bernhard Schätz and Christian Pfaller. Integrating Component Tests to System Tests. *Electronic Notes in Theoretical Computer Science (to appear)*.

[TTC] ETSI European Standard (ES) 201 873-1 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute, Sophia-Antipolis, 2007.

[WHR08] Matthias Wiese, Günter Hetzel, and Hans-Christian Reuss. Optimierung von E/E-Funktionstests durch Homogenisierung und Frontloading. In *AutoTest 2008*, Stuttgart, Germany, 2008.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol Specifications and Component Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.

[ZNZ08] Justyna Zander-Nowicka (Zander). *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*. PhD thesis, Technische Universität Berlin, 2008.

Der Einsatz quantitativer Sicherheitsanalysen für den risikobasierten Test eingebetteter Systeme[1]

Heiko Stallbaum, Andreas Metzger, Klaus Pohl

Software Systems Engineering
Universität Duisburg-Essen
Schützenbahn 70, 45127 Essen
{heiko.stallbaum|andreas.metzger|klaus.pohl}@sse.uni-due.de

Abstract: Um die oft hohen Sicherheitsanforderungen eingebetteter Systeme (im Sinne von Safety) zu gewährleisten, werden bei ihrer Entwicklung sowohl dynamische Testverfahren als auch statische Sicherheitsanalysen eingesetzt. Für den Test bieten sich insbesondere risikobasierte Ansätze an, da sie sicherstellen, dass solche Teile bzw. Anforderungen des Testobjektes früher und intensiver getestet werden, deren Versagen bzw. Nichterfüllung zu einem hohen Produktrisiko führen, also z.B. zu sehr hohen Schäden für Mensch und Umwelt. Der Einsatz eines risikobasierten Testansatzes erfordert jedoch aufgrund der notwendigen Produktrisikobewertungen hohen Zusatzaufwand. In diesem Beitrag wird ein Ansatz dargestellt, der risikobasierte Testtechniken mit quantitativen Sicherheitsanalysen kombiniert. Hierdurch werden Synergieeffekte erzielt, die eine separate Produktrisikobewertung obsolet machen und den entsprechenden Zusatzaufwand für den risikobasierten Test vermeiden. Die Anwendbarkeit des Ansatzes wird anhand eines Beispieles aus dem industriellen Umfeld demonstriert.

1 Einleitung und Motivation

Eingebettete Systeme unterliegen oftmals hohen Sicherheitsanforderungen (im Sinne von Safety). Solche sicherheitsgerichteten, eingebetteten Systeme, wie sie z.B. im Automobilbau oder der Luft- und Raumfahrt verwendet werden, bedürfen vor dem Einsatz der Zertifizierung, da sie zu Gefährdungen für Mensch und Umwelt führen können. Im Rahmen der Zertifizierung ist der Nachweis zu erbringen, dass schon im Entwicklungsprozess geeignete Maßnahmen ergriffen wurden, die sicherstellen, dass alle Sicherheitsanforderungen eingehalten werden. Zu diesen Maßnahmen zählen insbesondere dynamische Testverfahren und statische Sicherheitsanalysen (vgl. [Tr99], [Li02]).

Das primäre Ziel von Sicherheitsanalysen besteht darin, potenzielle Sicherheitsprobleme frühzeitig, vorausschauend aufzuspüren und zu gewichten (vgl. [Li02]). Für die Zertifizierung sicherheitsgerichteter, eingebetteter Systeme fordern Sicherheitsnormen die Anwendung ausgereifter, normierter Techniken der Sicherheitsanalyse (z.B. FMEA, FMECA, FTA, etc.) um die Sicherheit eines Systems nachzuweisen (vgl. z.B. [SAE96],

[1] Dieser Arbeit wurde mit Mitteln des Bundesministeriums für Bildung und Forschung (BMBF) unter dem Förderkennzeichen O1IS08045V (Softwareplattform Embedded Systems 2020, SPES 2020) gefördert.

[DIN03], [ISO09]). Hierbei wird zunächst ein Systemmodell erstellt und dieses dann systematisch hinsichtlich der Risiken analysiert. Dabei kommen sowohl qualitative als auch quantitative Analysen zum Einsatz. Sicherheitsanalysen bieten den Vorteil generelle Aussagen zur Sicherheit eines analysierten Systems zu liefern, da sie nicht wie dynamische Tests auf stichprobenartige Testfälle beschränkt sind. Die Sicherheitsanalyse hat jedoch den Nachteil, dass die Verlässlichkeit der gewonnenen Aussagen stark von der Übereinstimmung zwischen dem Systemmodell und der Realität abhängt (vgl. [Li02]). Gründe für eine mangelnde Übereinstimmung zwischen Modell und Realität können aus fehlerhafter Modellierung oder einer zu starke Abstraktion vom System mit entsprechendem Informationsverlust resultieren (vgl. [Tr99]).

Testen ist die vorherrschende Qualitätssicherungsmaßnahme in der Praxis. Tests haben den Vorteil, dass die zu testende Software in ihrer realen Betriebsumgebung ausgeführt werden kann, was die Aufdeckung von Fehlern oder Sicherheitsproblemen gestattet, die durch Sicherheitsanalysen kaum erkennbar sind (vgl. [Li02]). Durch Testen kann man Software jedoch nur stichprobenartig prüfen, da ein erschöpfender Test aufgrund der sehr großen Menge möglicher Eingaben nur in trivialen Fällen möglich ist. Ein systematisches Vorgehen beim Test kann sicherstellen, dass mit den gewählten Stichproben die wesentlichen Aspekte eines Softwaresystems getestet werden. Bei sicherheitsgerichteten, eingebetteten Systemen ist es daher wesentlich, dass diejenigen Teile bzw. Anforderungen getestet werden, deren Versagen bzw. Nichterfüllung zu einem hohen Produktrisiko führen, also z.B. zu sehr hohen Schäden für Mensch und Umwelt. Hier bieten sich risikobasierte Testansätze an, da diese sicherstellen, dass solche „riskanten“ Teile bzw. Anforderungen früher und intensiver getestet werden und damit die Wahrscheinlichkeit erhöht wird, dass kritische Fehler frühzeitig aufgedeckt werden.

1.1 Problemstellung

Der Einsatz eines risikobasierten Testansatzes erfordert eine Produktrisikobewertung. Deren Güte hat maßgeblichen Einfluss auf die Allokation des Testaufwandes und somit auf die Wahrscheinlichkeit des frühzeitigen Aufdeckens kritischer Fehler. Dementsprechend hoch sind die Anforderungen an die Risikobewertung: Sie muss systematisch alle Produktrisiken identifizieren und bewerten. Insbesondere in iterativen Entwicklungsprozessen mit veränderlichen Produktrisiken müssen Produktrisikobewertungen kontinuierlich aktualisiert werden, da veraltete Bewertungen zu einer falschen Allokation des Testaufwandes führen können (vgl. [Bo88], [Ba99], [Pi04]). Dieser Beitrag adressiert das Problem, dass beim risikobasierten Test hoher Zusatzaufwand für kontinuierlich aktualisierte und qualitativ gute Produktrisikobewertungen erforderlich wird.

1.2 Lösungsidee

Sowohl bei Sicherheitsanalysen als auch bei risikobasierten Tests werden Produktrisiken explizit berücksichtigt. In diesem Beitrag wird ein Ansatz vorgestellt, der risikobasierte Testtechniken mit quantitativen Sicherheitsanalysen kombiniert und hierdurch Synergieeffekte realisiert, durch die der in Abschnitt 1.1 dargestellte Zusatzaufwand für den risikobasierten Test vermieden wird. Dies wird erreicht, indem Produktrisikobewertungen

aus quantitativen Sicherheitsanalysen beim anforderungsbasierten Systemtest sicherheitsgerichteter, eingebetteter Systeme zur modell- und risikobasierten Testfallgenerierung und -priorisierung verwendet werden. Gegenüber existierenden risikobasierten Testansätzen bietet der vorgeschlagene Ansatz folgende Vorteile:

1. *Technik für Produktrisikobewertung*: Anstelle einer ansatzspezifischen Technik zur Produktrisikobewertung wird eine ausgereifte, normierte Technik der quantitativen Sicherheitsanalyse verwendet, deren Anwendung von Sicherheitsnormen für die Zertifizierung sicherheitsgerichteter, eingebetteter Systeme empfohlen wird. Dieses Vorgehen verspricht qualitativ gute und normenkonforme Produktrisikobewertungen sowie einen einfachen Transfer des vorgestellten Ansatzes in die Praxis.
2. *Aufwand für Produktrisikobewertung*: Separate Produktrisikobewertungen für den risikobasierten Test werden obsolet, da auf die Produktrisikobewertungen aus der Sicherheitsanalyse zurückgegriffen wird. Dieses Vorgehen verspricht eine Aufwandsvermeidung gegenüber Ansätzen mit separater Produktrisikobewertung.
3. *Aktualisierung von Produktrisikobewertungen:* Typischerweise wird eine prozessbegleitende Durchführung von Sicherheitsanalysen während der Entwicklung gefordert (vgl. [DIN03], [DIN06], [DT08]). Dementsprechend kann im vorgeschlagenen Ansatz auf kontinuierlich aktualisierte Risikobewertungen zurückgegriffen werden.

2 Verwandte Arbeiten

Existierende risikobasierte Testansätze geben entweder keine Anleitung zur Ausgestaltung der Produktrisikobewertung (vgl. [CP03]) oder schlagen jeweils ansatzspezifische Produktrisikobewertungen vor. Hierbei identifizieren und bewerten Experten oder Stakeholder Produktrisiken in Workshops z.B. auf Basis von Checklisten, Qualitätsmerkmalen und Priorisierungsmodellen (vgl. [Pi04]), gewichteter Risikofaktoren (vgl. [SW05], [Sr08]) oder mittels Heuristiken (vgl. [Ba99]). Einige Ansätze schlagen Metriken vor, um den durch Produktrisikobewertungen resultierenden Zusatzsaufwand für den risikobasierten Test zumindest zu reduzieren (vgl. [Ro99], [SM07]).

Liggesmeyer unterstreicht in [Li02] die bedeutende Rolle des Testens bei der Prüfung sicherheitsgerichteter, eingebetteter Systeme. Sicherheitsanalysen sollten ergänzend hierzu durchgeführt werden. Unter anderem wird vorgeschlagen die FMECA (Failure Mode, Effects and Criticality Analysis, [DIN06]) zumindest für das zu entwickelnde System und die kritischen Komponenten durchzuführen. Dabei erkannte schwerwiegende Risiken sollten mittels FTA (Fault Tree Analysis, [DIN07]) genauer quantitativ bestimmt werden. Die Integration oder Verwendung von (quantitativen) Sicherheitsanalysen in (risikobasierten) Testansätzen wird in [Li02] jedoch nicht adressiert.

Tracey et al. schlagen in [Tr99] die Integration eines Ansatzes zur automatischen Testdatengeneration in die Sicherheitsanalyse vor. Zentraler Bestandteil des Ansatzes ist eine Kostenfunktion, mit der Testdaten gesucht werden, die beim Test mit hoher Wahrscheinlichkeit zu einer Verletzung einer Sicherheitsanforderung führen. Diese Testdaten konzentrieren sich auf Teile des Softwaresystems, deren Sicherheit noch nicht hinreichend analysiert wurde. Bei der Sicherheitsanalyse mittels FTA werden die Testergebnisse

dann verwendet, um ein adressiertes Sicherheitsproblem nachzuweisen oder Vertrauen darin zu schaffen, dass es nicht vorliegt und die entsprechende Sicherheitsanforderung damit erfüllt ist. Dieses Vorgehen soll zu einer Aufwandsreduktion bei der Sicherheitsanalyse führen und das Vertrauen in den Sicherheitsnachweis stärken. Der Ansatz von Tracey et al. nutzt somit Testergebnisse für die Sicherheitsanalyse, beschreibt jedoch nicht, wie in umgekehrter Richtung (quantitative) Sicherheitsanalysen für den (risikobasierten) Test genutzt werden können.

In den Vorarbeiten der Autoren in [Ba08] wurde ein Ansatz zur risikobasierten Ableitung und Priorisierung von Testfällen für den modellbasierten Systemtest vorgeschlagen. Er erlaubt eine weitgehend automatische Generierung und Priorisierung von Testfällen auf Basis eines zustandsbasierten Testmodells ausgehend von risikobewerteten Anwendungsfällen. Zimmermann et al. stellen in [Zi09] aufbauend auf [Ba08] einen Ansatz zur automatisierten, risikobasierten Testfallgenerierung auf Basis statistischer Testmodelle für sicherheitskritische Systeme vor. Zur Risikobewertung sollen Sicherheitsanalysetechniken wie HAZOP (Hazard and Operability Study, [Ea92]), FMEA oder FTA eingesetzt werden. Es werden aber keine Hinweise gegeben, wie die Integration dieser Sicherheitsanalysen in die risikobasierte Testfallgenerierung erfolgen soll.

3 Ansatz für den risikobasierten Test unter Verwendung quantitativer Sicherheitsanalysen

Der in diesem Beitrag vorgeschlagene Ansatz adressiert zwei Phasen des modell- und risikobasierten Tests: (1) Die manuelle Testmodellerstellung sowie (2) die automatische Testfallableitung und -priorisierung (Abbildung 1). Im Gegensatz zu existierenden risikobasierten Testansätzen werden bei der Testmodellerstellung Produktrisikobewertungen aus quantitativen Sicherheitsanalysen verwendet (Abschnitt 3.1). Durch dieses Vorgehen kann der in Abschnitt 1.1 dargestellte Zusatzaufwand für den risikobasierten Test vermieden werden. Die im Rahmen der Sicherheitsanalyse quantifizierten Produktrisiken werden dann zur automatischen Testfallableitung und Priorisierung genutzt (Abschnitt 3.2). Die Aktivitäten der Phasen werden im Folgenden näher beschrieben.

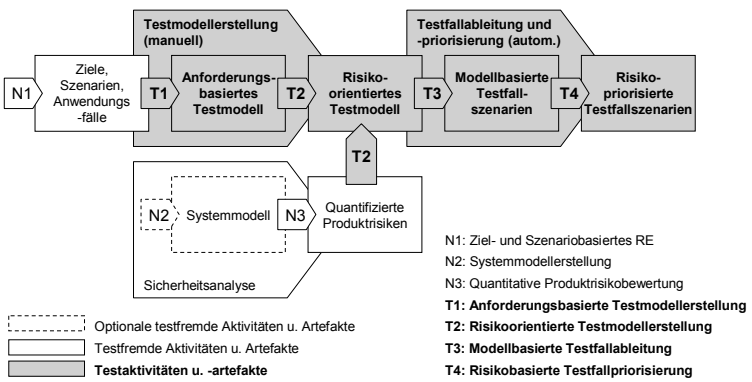


Abbildung 1: Anforderungs- und modellbasierter Ansatz für den risikobasierten Systemtest

3.1 Manuelle Testmodellerstellung unter Verwendung von Sicherheitsanalysen

In der ersten Phase des vorgeschlagenen Ansatzes wird ein sogenanntes risikoorientiertes Testmodell erstellt. Zur Erstellung dieses Modells sind quantifizierte Produktrisiken erforderlich. Bei der Entwicklung eingebetteter Systeme werden solche Risiken durch ausgereifte, normierte Techniken der quantitativen Sicherheitsanalyse ermittelt. Da Sicherheitsanalysen typischerweise prozessbegleitend durchgeführt werden, kann auf kontinuierlich aktualisierte Risikobewertungen zurückgegriffen werden. In diesem Abschnitt wird gezeigt, wie Aktivitäten der Sicherheitsanalyse und der Testmodellerstellung kombiniert werden können, damit separate, ansatzspezifische Produktrisikobewertungen für den risikobasierten Test sowie deren Aktualisierungen überflüssig werden.

Aktivität N1 (Ziel- und Szenariobasiertes Requirements Engineering): Anforderungen in Form von Anwendungsfällen bilden die Grundlage für den vorgeschlagenen Ansatz. Anwendungsfälle eignen sich zur Dokumentation von Anforderungen und als Grundlage für Sicherheitsanalysen (vgl. [AK01]). Sie aggregieren mehrere zu einem Ziel assoziierte Szenarien (vgl. [Po08]). Der vorgeschlagene Ansatz basiert somit auf Zielen, Szenarien und Anwendungsfällen, welche zentrale Ergebnisse eines ziel- und szenariobasierten Requirements Engineering darstellen.

Aktivität T1 (Anforderungsbasierte Testmodellerstellung): Als Zwischenschritt zum risikoorientierten Testmodell wird aus Anwendungsfällen zunächst ein anforderungsbasiertes Testmodell erstellt. In [Re05] und [Po08] wird mit ScenTED eine Technik dargestellt, mit der u.a. ausgehend von Anwendungsfällen und deren Szenarien das Gesamtverhalten eines Softwaresystems in einem Testmodell beschrieben werden kann. Die ScenTED-Technik verwendet zur Dokumentation des Testmodells UML Aktivitätsdiagramme. Ein wesentlicher Nutzen, der bei ScenTED durch die systematische Testmodellerstellung aus Anwendungsfällen erzielt wird, ist die Nachvollziehbarkeit (engl. Traceability) zwischen den in Anwendungsfällen aggregierten Zielen und Szenarien und Modellelementen des Testmodells (vgl. [Po08]). Im vorgeschlagenen Ansatz erfolgt die anforderungsbasierte Testmodellerstellung entsprechend der ScenTED-Technik.

Aktivität N2 (Systemmodellerstellung): Bei der Sicherheitsanalyse muss das zu analysierende System zunächst modelliert werden. Da das anforderungsbasierte Testmodell das Gesamtverhalten des zu testenden Softwaresystems darstellt, kann es bei der Sicherheitsanalyse als Systemmodell verwendet werden. In diesem Fall ergibt sich neben dem in diesem Beitrag adressierten Synergieeffekt für den risikobasierten Systemtest ein zusätzlicher Vorteil für die Sicherheitsanalyse: Durch Rückgriff auf das anforderungsbasierte Testmodell kann auf eine separate Systemmodellierung für die Sicherheitsanalyse verzichtet werden. Der vorgeschlagene Ansatz erlaubt auch die Verwendung quantitativer Sicherheitsanalysen, die auf anderen Systemmodellen als dem anforderungsbasierten Testmodell beruhen. In diesem Fall ist die Nachvollziehbarkeit zwischen den Modellelementen des verwendeten Systemmodells und den in Anwendungsfällen aggregierten Zielen und Szenarien notwendig, da sich die bei der Sicherheitsanalyse gewonnenen quantifizierten Produktrisiken auf Elemente des Systemmodells beziehen und daher bei der risikoorientierten Testmodellerstellung (Aktivität T2) zu den Elementen des Testmodells assoziiert werden müssen.

Aktivität N3 (Quantitative Produktrisikobewertung): Im Rahmen der quantitativen Sicherheitsanalyse werden Produktrisiken nunmehr auf Grundlage des Systemmodells und in Kenntnis der in Anwendungsfällen aggregierten Ziele und Szenarien systematisch analysiert und quantifiziert. In diesem Beitrag schlagen wir hierfür mit FMECA eine ausgereifte und normierte Technik der quantitativen Sicherheitsanalyse vor. Bei FMECA handelt es sich im Grunde um eine auf Risiken fokussierte Inspektionstechnik (vgl. [Li02]). Während der FMECA werden zunächst mögliche Fehlerarten identifiziert, hierfür dann u.a. Fehlerursachen und -auswirkungen analysiert sowie schließlich Risiken bestimmt, die auf einer Ordinalskala in eine Reihenfolge gebracht werden. Für die Entwicklung sicherheitsgerichteter, eingebetteter Systeme existieren zahlreiche Sicherheitsnormen, welche die Anwendung der FMECA für die Zertifizierung empfehlen (vgl. z.B. [SAE96], [DIN03], [ISO09]). Neben FMECA existieren weitere Techniken der Sicherheitsanalyse. Manche von ihnen sind stärker mathematisch fundiert, jedoch auch komplexer und weniger intuitiv anwendbar (Markov-Ketten, Petrinetze, etc.). Sie sind in der Praxis entsprechend weniger verbreitet und akzeptiert, können intuitiv anwendbare Techniken wie FMECA jedoch ergänzen (vgl. [DT08]). Die Wahl von FMECA verspricht somit qualitativ gute und normenkonforme Produktrisikobewertungen sowie einen einfachen Transfer des vorgestellten Ansatzes in die Praxis.

Aktivität T2 (Risikoorientierte Testmodellerstellung): Die im Rahmen der Sicherheitsanalyse gewonnenen quantifizierten Produktrisiken müssen explizit dokumentiert und in einem ausreichenden Umfang formalisiert werden. Im vorgeschlagenen Ansatz erfolgt die Dokumentation in einem zentralen intermediären Artefakt, dem risikoorientierten Testmodell. Es bildet die Grundlage für die weiteren Testaktivitäten wie etwa der modellbasierten Testfallableitung (Aktivität T3) oder der risikobasierten Testfallpriorisierung (Aktivität T4). Beide Aktivitäten können auf dieser Grundlage vollständig automatisiert durchgeführt werden. Das risikoorientierte Testmodell erweitert das anforderungsbasierte Testmodell um eine explizite Dokumentation der quantifizierten Produktrisiken. Als formale Notation wird aufbauend auf vorherigen Arbeiten das UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (UML QoS-Profil, [OMG08]) genutzt und eine QoS-Characteristic Risiko definiert (vgl. [Ba08]). Jede Aktualisierung einer Produktrisikobewertung aufgrund der prozessbegleitenden Durchführung von Sicherheitsanalysen erfordert lediglich die Aktualisierung des entsprechenden Produktrisikos im Testmodell.

3.2 Automatische Testfallableitung und -priorisierung

Das risikoorientierte Testmodell bildet die Grundlage für die zweite Phase des modell- und risikobasierten Tests: Die automatische Testfallableitung und -priorisierung. In [St08] haben wir mit RiteDAP eine Technik zur modellbasierten Testfallableitung und risikobasierten Testfallpriorisierung vorgestellt, die in Zusammenarbeit mit Industriepartnern im Rahmen des Forschungsprojektes ranTEST entwickelt wurde. Die RiteDAP-Technik verwendet zur Ableitung der Testfälle ein Testmodell, welches dem in Abschnitt 3.1 beschriebenen risikoorientierten Testmodell des vorgeschlagenen Ansatzes entspricht. Ein wesentlicher Nutzen, der bei RiteDAP durch die modellbasierte Testfallableitung und die explizite Dokumentation quantifizierter Produktrisiken im Testmodell

erzielt wird, ist der hohe Automatisierungsgrad (vgl. [St08]). Die umfassende Automatisierung der RiteDAP-Technik ermöglicht eine kosteneffiziente und zuverlässige Testfallableitung und -priorisierung beim risikobasierten Test, selbst wenn kontinuierliche Produktrisikobewertungen aufgrund veränderlicher Produktrisiken erforderlich sind (vgl. Abschnitt 1.1). Im vorgeschlagenen Ansatz erfolgt die Testfallableitung und -priorisierung entsprechend der RiteDAP-Technik.

Aktivität T3 (Modellbasierte Testfallableitung): Zur modellbasierten Ableitung von Testfällen werden bei RiteDAP automatisch Pfade durch das Testmodell bestimmt. Diese Pfade stellen Testfallszenarien dar, die von Testdaten abstrahieren. Die Auswahl der Testfallszenarien aus der Menge aller möglichen Testfallszenarien wird bei RiteDAP mit Hilfe des Boundary-Interior-Pfadüberdeckungskriteriums gesteuert. Testfallszenarien werden so abgeleitet, dass eine 100%-ige Abdeckung aller Pfade unter Berücksichtigung des Boundary-Interior-Kriteriums im risikoorientierten Testmodell erreicht wird.

Aktivität T4 (Risikobasierte Testfallpriorisierung): Die im risikoorientierten Testmodell annotierten quantifizierten Produktrisiken werden bei RiteDAP zur Priorisierung der abgeleiteten Testfallszenariomenge herangezogen. Hierzu wird für jedes Testfallszenario ein Risikowert errechnet. Der Risikowert ergibt sich aus der Summe der Risikowerte aller Aktivitäten, die das Testfallszenario im risikoorientierten Testmodell abdeckt (vgl. [CP03]). Je nach gewählter Priorisierungsstrategie werden hierbei schon abgedeckte Aktionen in die Summierung einbezogen (Total Risk Score Prioritization, TRSP) oder nicht (Additional Risk Score Prioritization, ARSP). Beide Priorisierungsstrategien sind Adaptionen grundlegender Priorisierungsstrategien aus [RE03] für den und modell- und risikobasierten Test auf Basis von Aktivitätsdiagrammen und werden in [St08] detailliert beschrieben.

4. Anwendung des Ansatzes

Zur Demonstration der Anwendbarkeit des Ansatzes wurde mit der adaptiven Fahrgeschwindigkeitsregelung bei Automobilen (engl. Adaptive Cruise Control, ACC) ein bekanntes sicherheitsgerichtetes, eingebettetes System aus dem industriellen Umfeld als Anwendungsbeispiel gewählt. Das ACC-System ist eine Weiterentwicklung der konventionellen Fahrgeschwindigkeitsregelung, die bei der Regelung den Abstand zu einem vorausfahrenden Fahrzeug als zusätzliche Stellgröße einbezieht. Das ACC-System reagiert auf langsamer vorausfahrende oder einscherende Fahrzeuge mit einer Reduzierung der eigenen Fahrgeschwindigkeit, so dass der vorgeschriebene Sicherheitsabstand zum vorausfahrenden Fahrzeug nicht unterschritten wird. Das im Folgenden beschriebene System basiert auf der Spezifikation des ACC-Systems nach [Wi03]. Es enthält auf Gesamtsystemebene insgesamt 16 Anwendungsfälle. Mit dem Ziel, die Anschaulichkeit und Verständlichkeit der Anwendbarkeitsdemonstration zu erhöhen, wurden technische Details des ACC-Systems zum Teil stark vereinfacht und die Betrachtung des ACC-Systems wurde auf einen einzelnen beispielhaften Anwendungsfall begrenzt. Im Folgenden werden die Ergebnisse der Aktivitäten des Ansatzes (s. Abschnitt 3) für diesen beispielhaften Anwendungsfall dargestellt. Eine weitergehende Validierung ist Gegenstand zukünftiger Arbeiten im Rahmen der nationalen Innovationsallianz SPES 2020.

Aktivität N1 (Ziel- und Szenariobasiertes Requirements Engineering): Als Ergebnis des ziel- und szenariobasierten Requirements Engineering dokumentiert Tabelle 1 den betrachteten Anwendungsfall mit Hilfe einer Schablone (vgl. [Po08]). Der Anwendungsfall gruppiert jene Szenarien, die das ACC-System ausführt, um das Ziel „Abstand zum vorausfahrenden Fahrzeug vergrößern um Sicherheitsabstand einzuhalten" zu erreichen.

Abschnitt	Inhalt
Name	Geschwindigkeitsverringerung bei vorausfahrendem Fahrzeug innerhalb des Sicherheitsabstandes
Ein- und Ausgabevariablen	▪ Input: Abstand zum vorausfahrenden Fahrzeug von Abstandssensoren des ACC-Systems, Ist-Geschwindigkeit von Tachometer ▪ Output: Signal zur Geschwindigkeitsverringerung an Motorsteuerung, ggf. Bremssignal an Bremssystem
Ziel(e)	Abstand zum vorausfahrenden Fahrzeug vergrößern um Sicherheitsabstand einzuhalten
Primärer Akteur	Vorausfahrendes Fahrzeug
Andere Akteure	Tachometer, Motorsteuerung, Bremssystem
Vorbedingungen	ACC-System wurde mit Soll-Geschwindigkeit aktiviert
Nachbedingungen	Geschwindigkeit wurde auf Folgegeschwindigkeit verringert
Auslöser	Abstand zum vorausfahrenden Fahrzeug ist zu gering
Hauptszenario	1. ACC-System errechnet Folgegeschwindigkeit auf Grundlage von Ist-Geschwindigkeit und Abstand zum vorausfahrenden Fahrzeug 2. ACC-System sendet Signal zur Geschwindigkeitsverringerung auf Folgegeschwindigkeit an Motorsteuerung
Alternativszenarien	2a1. ACC-System stellt fest, dass Geschwindigkeitsverringerung durch Motorsteuerung nicht ausreichend ist 2a2. ACC-System sendet Bremssignal auf Folgegeschwindigkeit an Bremssystem

Tabelle 1: Betrachteter Anwendungsfall des ACC-Systems

Aktivität T1 (Anforderungsbasierte Testmodellerstellung): Auf Basis der Anwendungsfälle eines Systems und insbesondere auf Basis der in ihnen enthaltenen Szenarien wird mittels der ScenTED-Technik ein anforderungsbasiertes Testmodell erstellt, welches als Aktivitätsdiagramm dokumentiert wird. Mit ScenTED wird die Nachvollziehbarkeit zwischen den in Anwendungsfällen aggregierten Zielen und Szenarien und Modellelementen des Testmodells gewährleistet. Abbildung 2 stellt den Ausschnitt des anforderungsbasierten Testmodells für das ACC-System dar, welcher das Verhalten des betrachteten Anwendungsfalls modelliert.

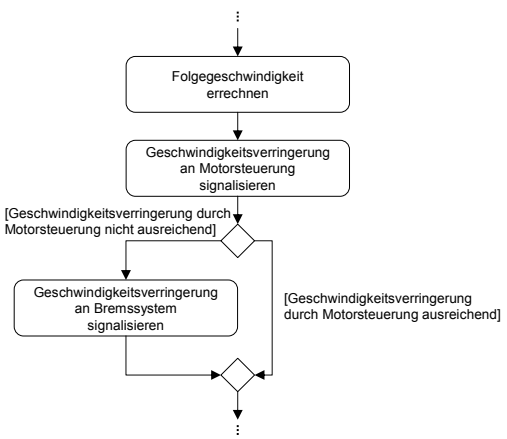


Abbildung 2: Ausschnitt aus anforderungsbasiertem Testmodell zum betrachteten Anwendungsfall

Aktivität N2 (Systemmodellerstellung): Im Rahmen der quantitativen Sicherheitsanalyse kann das anforderungsbasierte Testmodell als Systemmodell verwendet werden. Wir

demonstrieren im Folgenden die Verwendung dieses Modells bei der quantitativen Sicherheitsanalyse, da hierdurch eine separate Systemmodellierung obsolet wird.

Aktivität N3 (Quantitative Produktrisikobewertung): Sicherheitsanalysen (z.B. durch FMECA) werden von Sicherheitsstandards für eingebettete Systemen im Automobil gefordert (vgl. [ISO09]). Im Rahmen der quantitativen Sicherheitsanalyse mit FMECA werden mögliche Fehlerarten zum ACC-System identifiziert, zugehörige Fehlerursachen und -auswirkungen analysiert sowie Produktrisiken quantifiziert. Die Ergebnisse der FMECA werden in FMECA-Arbeitsblättern dokumentiert. In Tabelle 2 ist ein Teil des Analyseergebnisses der FMECA für den betrachteten Anwendungsfall in einem solchen FMECA-Arbeitsblatt dargestellt. Die Risikoprioritätszahl RPN dient zur Quantifizierung der identifizierten Produktrisiken und wurde im Beispiel als Produkt der Schwere der Fehlerauswirkung und des Auftretens bzw. der Auftretenswahrscheinlichkeit der Fehlerursachen berechnet.[2] Hierbei wurden die in der Automobilindustrie weit verbreiteten Skalen für Schwere und Auftreten von 1 bis 10 (sehr gering = „1“ bis sehr hoch = „10“) verwendet (vgl. [DIN06]).

Tabelle 2: Ausschnitt aus FMECA-Arbeitsblatt zum betrachteten Anwendungsfall

Funktion / Anforder.	Fehlerart(en)	Fehlerauswrikung(en)	Schwere	Fehlerursache(n)	Auftreten	RPN
Folgegeschwindigkeit errechnen	Folgegeschwindigkeit kann nicht errechnet werden	Fahrzeug wird nicht verlangsamt und kollidiert mit vorausfahrendem Fahrzeug	10	Ist-Geschwindigkeit und / oder Abstand zum vorausfahrenden Fahrzeug liegen nicht vor	1	10
			10	Softwarefehler bei der Berechnung	2	20
	Folgegeschwindigkeit wird zu langsam errechnet	Fahrzeug wird verspätet verlangsamt und kollidiert mit vorausfahrendem Fahrzeug	10	Ist-Geschwindigkeit und / oder Abstand zum vorausfahrenden Fahrzeug liegen verspätet vor	1	10
			10	Softwarefehler bei der Berechnung	2	20
		Fahrzeug wird verspätet verlangsamt und kann Sicherheitsabstand nicht einhalten	4	Ist-Geschwindigkeit und / oder Abstand zum vorausfahrenden Fahrzeug liegen verspätet vor	1	4
			4	Softwarefehler bei der Berechnung	2	8
	Folgegeschwindigkeit wird zu hoch errechnet	Fahrzeug wird zu wenig verlangsamt und kollidiert mit vorausfahrendem Fahrzeug	10	Ist-Geschwindigkeit liegt falsch (zu niedrig) und / oder Abstand zum vorausfahrenden Fahrzeug falsch (zu groß) vor	3	30
			10	Softwarefehler bei der Berechnung	2	20
	Folgegeschwindigkeit wird zu niedrig errechnet	Fahrzeug wird stärker verlangsamt als zur Einhaltung des Sicherheitsabstandes notwendig	1	Ist-Geschwindigkeit liegt falsch (zu hoch) und / oder Abstand zum vorausfahrenden Fahrzeug falsch (zu klein) vor	3	3
			1	Softwarefehler bei der Berechnung	2	2

Aktivität T2 (Risikoorientierte Testmodellerstellung): Aus dem anforderungsbasierten Testmodell zum ACC-System, welches im Beispiel auch das Systemmodell bei der FMECA ist, sowie aus den quantifizierten Produktrisiken der FMECA wird das risikoorientierte Testmodell gebildet. Im risikoorientierten Testmodell werden die analysierten Risiken explizit dokumentiert und formalisiert. Abbildung 3 stellt den Ausschnitt des risikoorientierten Testmodells für das ACC-System dar, welcher das Verhalten des betrachteten Anwendungsfalls modelliert. Den drei Aktivitäten des Anwendungsfalls wird als Risikowert jeweils die Summe aller Risikoprioritätszahlen aus dem FMECA-Arbeitsblatt zugewiesen, die sich auf Fehlerauswirkungen bzw. Fehlerursachen zur Aktivität beziehen (analog zur Summierung von Risiken verschiedener Fehlerereignisse in [Ro99]). Die Annotation der Risikowerte im Testmodell erfolgt auf Basis des UML

[2] Einige Anwendungen der FMECA unterscheiden zusätzlich den Grad der Fehlererkennung auf Systemebene. In diesen Anwendungen wird eine zusätzliche Kategorie für Fehlererkennung benutzt um die RPN als Produkt dreier Faktoren zu bilden.

QoS-Profils in Kommentaren zu den entsprechenden Testmodellelementen (Aktionen). Eine erneute Durchführung der FMECA im Rahmen der prozessbegleitenden Sicherheitsanalyse macht lediglich die Aktualisierung dieser Kommentare notwendig.

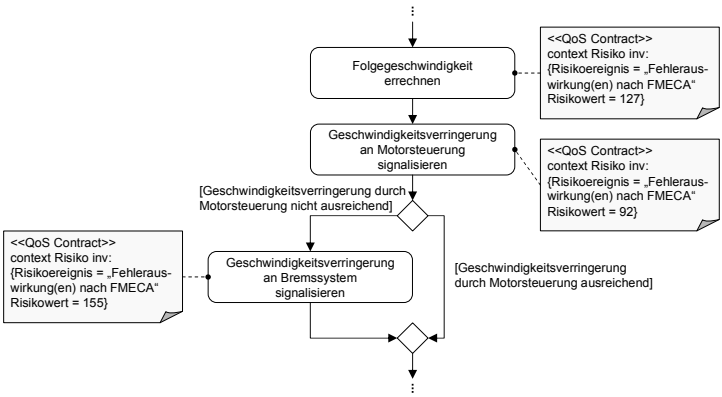


Abbildung 3: Ausschnitt aus risikoorientiertem Testmodell zum betrachteten Anwendungsfall

Aktivität T3 (Modellbasierte Testfallableitung): Das risikoorientierte Testmodell kann direkt zur automatischen Testfallableitung mittels RiteDAP genutzt werden. Tabelle 3 listet die beiden möglichen Pfade durch den Testmodellausschnitt aus Abbildung 3. Die Pfade stellen Teilpfade durch das risikoorientierte Testmodell des Gesamtsystems dar und somit Teilszenarien der für das Gesamtsystem ableitbaren Testfallszenarien für den Systemtest.

Aktivität T4 (Risikobasierte Testfallpriorisierung): In RiteDAP werden die abgeleiteten Testfälle automatisch priorisiert. Dies erfolgt auf Basis der im risikoorientierten Testmodell explizit dokumentierten quantifizierten Produktrisiken. In Tabelle 3 wird das Ergebnis dieser risikobasierten Testfallpriorisierung mit RiteDAP unter Verwendung der Priorisierungsstrategie TRSP für beide Testfallteilszenarien dargestellt.

Nummer	Testfallteilszenario	Risikowert
1	Folgegeschwindigkeit errechnen, Geschwindigkeitsverringerung an Motorsteuerung signalisieren, Geschwindigkeitsverringerung an Bremssystem signalisieren	374 (127+92+155)
2	Folgegeschwindigkeit errechnen, Geschwindigkeitsverringerung an Motorsteuerung signalisieren	219 (127+92)

Tabelle 3: Mögliche Testfallteilszenarien zum betrachteten Anwendungsfall und ihre Risikowerte

Das risikoorientierte Testmodell für das gesamte ACC-System enthält auf Gesamtsystemebene (16 Anwendungsfälle) ca. 110 Aktionen und ca. 30 Fallunterscheidungen. Die Vielzahl der hieraus ableitbaren Testfallszenarien macht die Notwendigkeit der automatischen Testfallpriorisierung deutlich. Ohne Testfallpriorisierung könnte eine Ordnung oder Auswahl der Testfallszenarien für die Testdurchführung nur zufällig getroffen werden. Und ohne Automatisierung der Testfallpriorisierung wäre mit jeder Aktualisierung der im risikoorientierten Testmodell dokumentierten Produktrisiken eine aufwändige und fehleranfällige manuelle Neupriorisierung aller Testfallszenarien notwendig.

5. Zusammenfassung und Ausblick

Dieser Beitrag hat gezeigt, wie sich Synergieeffekte durch die Nutzung quantitativer Sicherheitsanalysen für den risikobasierten Test realisieren lassen. Im vorgeschlagenen Ansatz werden quantifizierte Produktrisiken aus Sicherheitsanalysen beim risikobasierten Test sicherheitsgerichteter, eingebetteter Systeme zur automatischen Testfallableitung und -priorisierung genutzt. Hiermit wird der beim risikobasierten Test typischerweise anfallende Zusatzaufwand vermindert, da keine separaten Produktrisikobewertungen durchgeführt werden müssen. Insbesondere in iterativen Entwicklungsprozessen mit veränderlichen Produktrisiken ist eine fortlaufende Aktualisierung der Produktrisikobewertung für risikobasierte Tests notwendig, da veraltete Bewertungen zu einer falschen Allokation des Testaufwandes führen können. Da Sicherheitsanalysen während der Entwicklung prozessbegleitend durchgeführt werden, kann der vorgeschlagene Ansatz kontinuierlich aktualisierte Produktrisiken zur Testfallableitung und -priorisierung verwenden, was zu einer kosteneffizienten und zuverlässigen Allokation des Testaufwandes entsprechend der laufend aktualisierten Produktrisiken führt.

Mit der FMECA wurde in diesem Beitrag ein ausgereiftes und normiertes Verfahren der quantitativen Sicherheitsanalyse für die Produktrisikobewertung vorgeschlagen. Während die Anwendung der FMECA auf System- und Hardwareebene in der Praxis akzeptiert und weit verbreitet ist, wird Software bei der Sicherheitsanalyse eingebetteter Systeme vorwiegend als Blackbox betrachtet (vgl. [DT08]). Mit dem stetig wachsenden Anteil der durch Software realisierten Funktionalität eines eingebetteten Systems gewinnt die Sicherheitsanalyse der Software jedoch an Bedeutung. Oftmals wird der FMECA nur eine eingeschränkte Anwendbarkeit auf Software zugesprochen (vgl. [Li00]). Eine genaue Analyse der FMECA sowie weiterer Techniken der quantitativen Sicherheitsanalyse (z.B. FTA) hinsichtlich ihrer Eignung für den vorgeschlagenen Ansatz ist deshalb Gegenstand unserer zukünftigen Arbeiten.

Die FMECA wird erfolgreich vorzugsweise im frühen Entwicklungszyklus durchgeführt, damit die Behebung identifizierter Sicherheitsprobleme möglichst kosteneffizient ist. Sie ist ein iterativer Prozess, der den Entwicklungsprozess begleitet (vgl. [DIN06]). Üblicherweise wird die FMECA auf Wirkstrukturen durchgeführt, die sich aus Architekturen ergeben. Im vorgeschlagenen Ansatz haben wir gezeigt, wie in Anwendungsfällen aggregierte Ziele und Szenarien die Grundlage für die quantitative Sicherheitsanalyse bilden können und somit eine frühe, anforderungsbasierte FMECA ermöglicht wird (vgl. [AK01]). Es ist zu erwarten, dass zusätzliches Architekturwissen eine genauere Analyse ermöglicht und die Analyse normativen Anforderungen zudem besser gerecht wird. In SPES 2020 wird daher untersucht, wie sich Entwicklungsansätze, die den Architekturentwurf mit dem ziel- und szenariobasierten Requirements Engineering verzahnen (z.B. COSMOD-RE, [Po08]), positiv auf frühe Sicherheitsanalysen auswirken können.

Literaturverzeichnis

[AK01] K. Allenby, T. Kelly: Deriving Safety Requirements Using Scenarios. In: Proc. of 5th RE, Toronto, Ontario, Canada, 2001, pp. 228-235.

[Ba99] J. Bach: Risk-Based Testing. How to conduct heuristic risk analysis. In: Software Testing & Quality Engineering Magazine, vol. 1, iss. 6, 1999, pp. 23-28.
[Ba08] T. Bauer, H. Stallbaum, A. Metzger, R. Eschbach: Risikobasierte Ableitung und Priorisierung von Testfällen für den modellbasierten Systemtest. In: Proc. of SE, München, Germany, ser. LNI, vol. 121. GI, 2008, pp. 99-111.
[Bo88] B. W. Boehm: A spiral model of software development and enhancement. In: IEEE Computer, vol. 21, no. 5, pp. 61-72, 1988.
[CP03] Y. Chen, R.L. Probert: A Risk-based Regression Test Selection Strategy. In: Proc. of 14th ISSRE, Denver, CO, USA, 2003, pp. 305-306.
[DIN03] DIN EN 61508: Funktionale Sicherheit sicherheitsbezogener elektrischer / elektronischer / programmierbarer elektronischer Systeme. DIN, Germany, 2003.
[DIN06] DIN EN 60812: Analysetechniken für die Funktionsfähigkeit von Systemen – Verfahren für die Fehlzustandsart- und -auswirkungsanalyse (FMEA). DIN, Germany, 2006.
[DIN07] DIN EN 61025: Fehlerzustandsbaumanalyse. DIN, Germany, 2007.
[DT08] D. Domis, M. Trapp: Integrating Safety Analyses and Component-based Design. In: Proc. of 27th SAFECOMP, Newcastle upon Tyne, UK, 2008, pp. 58-71.
[Ea92] J. V. Earthy: Hazard and Operability Study as an Approach to Software Safety Assessment. In: Colloqium on Hazard Analysis, London, UK, 1992, pp. 5/1-5/3.
[ISO09] ISO/DIS 26262: Road vehicles – Functional safety. ISO, Switzerland, 2009.
[Li00] P. Liggesmeyer: Formale und stochastische Methoden zur Qualitätssicherung technischer Software (eingeladener Vortrag). In: Softwaretechnik-Trends, vol. 20, no. 3, 2000.
[Li02] P. Liggesmeyer: Software-Qualität. Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, 2002.
[OMG08] UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. Version 1.1. OMG, 2008.
[Pi04] I. Pinkster, B. van de Burgt, D. Janssen, and E. van Veenendaal: Successful Test Management: An Integral Approach, 2nd ed., Springer, Berlin, 2004.
[Po08] K. Pohl: Requirements Engineering: Grundlagen, Prinzipien, Techniken. 2nd ed., dpunkt Verlag, 2008.
[Re05] A. Reuys, E. Kamsties, K. Pohl, S. Reis: Model-based System Testing of Software Product Families. In: Proc. of 17th CAiSE, Porto, Portugal, 2005, pp. 519-534.
[RE03] G. Rothermel, S. Elbaum: Putting your best Tests forward. In: IEEE Software, vol. 20, no. 5, 2003, pp. 74-77.
[Ro99] L.H. Rosenberg, R. Stapko, A. Gallo: Risk-based Object Oriented Testing. In: Proc. of 24th annual Software Engineering Workshop, NASA, SEL, Greenbelt, MD, USA, 1999.
[SAE96] ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. SAE, 1996.
[SM07] H. Stallbaum, A. Metzger: Employing Requirements Metrics for Automating Early Risk Assessment. In: Proc. of MeReP07, Palma de Mallorca, Spain, 2007, pp. 1-12.
[Sr08] P. R. Srivastva, K. Kumar, and G. Raghurama: Test case prioritization based on requirements and risk factors. In: Softw. Eng. Notes, vol. 33, no. 4, 2008, pp. 1-5.
[St08] H. Stallbaum, A. Metzger, K. Pohl: An automated Technique for risk-based Test Case Generation and Prioritization. In: Proc. of 3rd AST, Leipzig, Germany, 2008, pp. 67-70.
[SW05] H. Srikanth, L. Williams: On the economics of requirements-based test case prioritization. In: Proc. of 7th EDSER, New York, NY, USA, 2005, pp. 1-3.
[Tr99] N. Tracey, J. Clark, J. Mcdermid, K. Mander: Integrating Safety Analysis with Automatic Test-data Generation for Software Safety Verification. In: Proc. of 17th ISSC, Orlando, FL, USA, 1999, pp. 128-137.
[Wi03] H. Winner: ACC Adaptive Cruise Control. Robert Bosch GmbH, Stuttgart, 2003.
[Zi09] F. Zimmermann, R. Eschbach, J. Kloos, T. Bauer: Risk-based statistical Testing: A refinement-based Approach to the Reliability Analysis of Safety-critical Systems. In: Proc of 12th EWDC, Toulouse, France, 2009.

Workshops

Enterprise Engineering meets Software Engineering (E²mSE)

Stefan Jablonski[1], Erich Ortner[2], Marco Link[2]

[1]Universität Bayreuth
stefan.jablonski@uni-bayreuth.de
[2]Technische Universität Darmstadt
{ortner, link}@winf.tu-darmstadt.de

Enterprise Engineering (EE) kann als die ganzheitliche Entwicklung und Implementierung eines Unternehmens unter Einsatz von IT verstanden werden. Damit verbunden ergeben sich Fragen zu Design, Modellierung, Flexibilität und internen sowie externen Interaktionen einer Organisation. Alle Anforderungen sind im Rahmen dessen auch auf die Abhängigkeiten der Unternehmensumwelt auszurichten. Innerhalb einer Institution (Architektur) kann als nächsthöhere Granularitätsstufe das Service Engineering gesetzt werden. Intern und extern zu verwendende Services sind gemäß vorliegenden Anforderungen und Vorgaben zu gestalten. Speziell im Bereich der Entwicklung und Anpassung von IT-Services oder anderen Softwarefragmenten ist die Disziplin des Software Engineerings (SE) gefragt. Ausgehend von diesem Aufbau stellt die Informationstechnologie eine wesentliche Basiskomponente der heutigen Unternehmenswertschöpfung dar. Mit all diesen und weiteren komplexen Herausforderungen hat sich das heutige und zukünftige Software Engineering auseinanderzusetzen. Neue Konzepte, Methoden und Werkzeuge sind zu diskutieren, zu analysieren und zu erforschen.

Erster Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme (ENVISION2020)

Manfred Broy[1], David Cruz[1], Martin Deubler[1],
Kim Lauenroth[2], Klaus Pohl[2], Ernst Sikora[2]

[1]Technische Universität München, Institut für Informatik I4,
Software & Systems Engineering, Boltzmannstraße 3, 85748 Garching
{broy, cruz, deubler}@in.tum.de
[2]Universität Duisburg-Essen, Institut für Informatik und Wirtschaftsinformatik,
Software Systems Engineering, Schützenbahn 70, 45127 Essen
{kim.lauenroth, klaus.pohl, ernst.sikora}@sse.uni-due.de

Softwareintensive, eingebettete Systeme unterstützen den Menschen schon heute in vielen Bereichen des Lebens – sichtbar und unsichtbar. Beispielsweise verbessern sie im Automobil die Sicherheit, regulieren das Klima in Gebäuden oder steuern medizinische Geräte bis hin zu ganzen Industrieanlagen. Experten prognostizieren für die Zukunft eine rasante Zunahme softwareintensiver, eingebetteter Systeme.

Die Ausweitung des Funktionsumfangs und die zunehmende Vernetzung eingebetteter Systeme führen gleichzeitig zu einer rasanten Zunahme der Komplexität dieser Systeme, die auch im Entwicklungsprozess Berücksichtigung finden muss. Existierende Vorgehensweisen und Methoden stoßen bereits unter den heutigen Rahmenbedingungen (z.B. Zeit- und Kostendruck) an ihre Grenzen. Existierende Ansätze und Methoden müssen aufgrund der wachsenden Herausforderungen in Frage gestellt und in Teilen neu konzipiert werden.

Der Workshop ENVISION 2020 verfolgt das Ziel, die Entwicklung und Diskussion zukünftiger Ansätze, Vorgehensweisen und Methoden zur Entwicklung softwareintensiver, eingebetteter Systeme zu fördern. Wir laden zu diesem Workshop Beiträge von Forschern und Praktikern ein, die diese Diskussion stimulieren und die Konzeption neuer, verbesserter Entwicklungsansätze mit gestalten wollen. Ein besonderes Augenmerk gilt dabei modellbasierten Entwicklungsansätzen.

Evolution von Software-Architekturen (EvoSA 2010)

Matthias Riebisch[1], Stephan Bode[1], Petra Becker-Pechau[2]

[1]Technische Universität Ilmenau
{matthias.riebisch|stephan.bode}@tu-ilmenau.de
[2]Universität Hamburg
becker@informatik.uni-hamburg.de

Softwaresysteme sind heute ständigen Anforderungen nach Änderungen ausgesetzt. Da die Softwaresysteme aus Kosten- und Zeitgründen nicht immer neu entwickelt werden können, ist ihre Anpassungsfähigkeit und Weiterentwickelbarkeit (Evolvability) über längere Zeit von entscheidender Bedeutung. Dies trifft gleichermaßen für Software-intensive Systeme zu. Softwarearchitekturen dienen als grundlegendes Beschreibungsmittel von Softwaresystemen und stellen damit den Ausgangspunkt zur Weiterentwicklung dar. Die Architekturevolution wird beeinflusst durch Prozesse, Aktivitäten und Beschreibungsmittel. Ihre Unterstützung durch Software Engineering Methoden ist heute jedoch noch unzureichend. Dieser Workshop hat das Ziel die deutschsprachige Community zusammenzubringen, um über aktuelle Arbeiten zu diskutieren und den Stand der Technik sowie die wichtigsten Forschungsfragen zu ermitteln. Zu diskutierende Aspekte sind beispielsweise der Architekturentwurf hinsichtlich Weiterentwickelbarkeit und Langlebigkeit von Software-Architekturen, statische und dynamische Aspekte der Weiterentwicklung von Architekturmodellen, die Rolle von aspektorientierter und modellgetriebener Architekturentwicklung, Evolvability als Qualitätsattribut von Software-Architekturen, Bewertungsmethoden und Metriken für die Evolvability von Architekturen oder Toolunterstützung beim Architekturdesign hinsichtlich Evolution.

3. Grid Workflow Workshop (GWW 2010)

Wilhelm Hasselbring[1], Andre Brinkmann[2]

[1]Universität Kiel
wha@informatik.uni-kiel.de
[2]Universität Paderborn
brinkman@uni-paderborn.de

Im Bereich des Grid Computing stehen Workflows im Fokus zahlreicher Projekte. Allein auf europäischer Ebene wurden und werden viele Projekte zur Entwicklung von geeigneten Werkzeugen, Sprachen und Laufzeitumgebungen für Workflows im Grid-Computing gefördert.

Der dritte Grid Workflow Workshop adressiert sowohl wissenschaftliche als auch betriebliche Workflows im Umfeld des Grid-Computing. Diese dritte Auflage legt den Schwerpunkt auf Fragen des Software Engineering für betriebliche und wissenschaftliche Workflows, Workflow- Sicherheitsinfrastrukturen und die Integration/Migration bestehender betrieblicher und Grid-spezifischer Infrastrukturen.

Die Themenfelder dieses Workshops:

- Kommerzielle und wissenschaftliche Grid Workflows
- Ausführungsumgebungen für Grid Workflows
- Grid Workflow-Sicherheit
- Workflows und Grid-Middleware
- EAI und Grid-Computing
- SOA mittels Grid-Technologien
- Grid Service Orchestrierung
- Scheduling und Workflows
- Workflow-Sprachen für Grid Workflows
- Domänenorientierte Grid Workflow-Definition
- Formale Modelle für die Grid Workflow-Analyse

3. Workshop zur Erhebung, Spezifikation und Analyse nichtfunktionaler Anforderungen in der Systementwicklung

Joerg Doerr, Peter Liggesmeyer

Fraunhofer-Institut für Experimentelles Software Engineering
Fraunhofer-Platz 1
67663 Kaiserslautern
{joerg.doerr, peter.liggesmeyer}@iese.fraunhofer.de

Die Beachtung von nichtfunktionalen Anforderungen (auch häufig als Qualitätsanforderungen bezeichnet) ist essentiell für erfolgreiche Projekte und Produkte. Neben der reinen Funktionalität finden Qualitätsattribute wie Effizienz, Benutzungsfreundlichkeit und Sicherheit in den verschiedensten Domänen Beachtung. Das Vernachlässigen dieser Qualitäten führt häufig zu gescheiterten Projekten, geringer Produktqualität, verlängerter TTM und hohem Rework-Aufwand. Erfolgreich umgesetzte Qualitätsanforderungen können Differenzierungsmerkmale der Produkte gegenüber Wettbewerbern darstellen. In der Praxis werden nichtfunktionale Anforderungen oftmals kaum oder nur ad hoc behandelt; eine systematische Erhebung, Spezifikation und Analyse konkreter und messbarer nichtfunktionaler Anforderungen ist oftmals nicht zu beobachten.

Der Workshop ist eine Fortsetzung des 1. und 2. Workshops auf der SE 2007 und SE 2008. Er bringt Praktiker und Akademiker, welche auf dem Gebiet der nichtfunktionalen Anforderungen arbeiten zu einem gemeinsamen Erfahrungsaustausch zusammen. Die Praktiker stellen im Workshop ihre aktuelle Verfahrensweise bzgl. nichtfunktionaler Anforderungen und Ihre aktuellen Herausforderungen dar. Aus akademischer Sicht werden aktuelle Lösungsansätze, Methodenbeschreibungen und aktuelle Forschungsthemen vorgestellt.

2nd European Workshop on Patterns for Enterprise Architecture Management (PEAM2010)

Florian Matthes, Sabine Buckl, Christian M. Schweda

Technische Universität München
{matthes, buckls, schweda}@in.tum.de

Abstract: Enterprise architecture (EA) and the holistic management thereof are topics of ongoing interest from practitioners, standardization bodies, and researchers. Not surprisingly, a large number of different approaches, frameworks, and guidelines for EA management have been developed in the last years, all targeting different aspects of the architecture and the corresponding management function, respectively. Therefore, a multitude of linguistic communities emerged around the subject, each using its distinct terminology as well as forms of presenting the approaches, frameworks, and guidelines. In this light, especially practitioners may find it increasingly complex to contribute their experience to the body of knowledge in the field of EA management.

EA management patterns (**EAM patterns**) form a technique to bring together practice-driven development and academic research. In the form of EAM patterns, both practitioners and researchers can identify, document, and exchange best practices for the management of EAs. An EAM pattern thereby describes a general, reusable solution to a common problem in a given organizational context. It identifies driving forces, known usages, and consequences. Such patterns can be specified on different levels of abstraction and detail, e.g. as a method for enterprise modeling, or a reference model for the EA management function. Furthermore, EAM patterns address social, technical, and economic issues in a balanced manner.

The PEAM workshop wants to provide a platform on which EAM best practices can be discussed and promulgated among European researchers and practitioners with experience in EA management and nearby topics. This is also reflected in the workshop's make-up consisting of a half-day "classical" workshop, during which research papers on EAM patterns are discussed, and a one-day pattern workshop. On this day, EAM patterns are subjected to intense discussions, where new ideas are collaboratively developed. With this twofold structure, the PEAM workshop brings together the advantages of classical paper-workshops and of pattern-workshops in the tradition of the pattern language conferences of the Hillside Group, as e.g. the PLoP®.

Topics: Reflecting the broadness of the field of EA management, the PEAM workshop targets manifold topics with an emphasis on reusable solutions for common problems in:

- Business-IT alignment
- Service-oriented design of enterprise architectures
- Modeling languages for enterprise architectures
- Metrics for application landscapes and enterprise architectures

- Modeling, simulation and monitoring of enterprise performance
- Usage of architectural blueprints and architectural standards
- Analysis of enterprise architectures
- Frameworks and methods for enterprise architecture management
- Enterprise architectures for the extended enterprise
- Social aspects of enterprise architecture management
- Tool support for enterprise architecture management
- Use of software design models for runtime application management
- Obtaining simplicity from enterprise architecture management
- Service-oriented combination of CSD and standard software packages
- Enterprise architecture management in transformation

Program Committee: The special format of the workshop and the utilization of patterns as preferred means of communication and presentation are also reflected in the program committee, which on the one hand brings together researchers and practitioners in the field of EA management, and on the other hand involves pattern experts with several years of PLoP®-experience. Each paper was reviewed by at least two members of the program committee, while papers that present patterns additionally underwent an extensive shepherding process to improve their quality and "patternness". The following people made this intensive reviewing and shepherding possible:

- Ademar Aguiar (Universidade do Porto, Portugal)
- Antonia Albani (Delft University of Technology, The Netherlands)
- Hans-Jürgen Appelrath (Universität Oldenburg, Germany)
- Gernot Dern (SEB Bank, Germany)
- Alexander Ernst (Technische Universität München, Germany)
- Ulrich Frank (Universität Duisburg Essen, Germany)
- Norbert Gronau (Universität Potsdam)
- Pontus Johnson (Royal Institute of Technology, Sweden)
- Paul Johannesson, (Royal Institute of Technology, Sweden)
- Dimitris Karagiannis (Universität Wien, Austria)
- Wolfgang Keller (objectarchitects, Germany)
- Marc Lankhorst (Novay, The Netherlands)
- Florian Matthes (Technische Universität München, Germany)
- Klaus D. Niemann (act! consulting, Germany)
- Erik Proper (Capgemini & Radboud University Nijmegen, The Netherlands)
- Ralf Reussner (Universität Karlsruhe, Germany)
- Michael Rohloff (Universität Potsdam, Germany)
- Peter Sommerlad (Hochschule Rapperswil, Switzerland)
- Ulrike Steffens (OFFIS, Germany)
- Johannes Willkomm (Capgemini sd&m Research, Germany)
- Robert Winter (Universität St. Gallen, Switzerland)
- Christian Winterhalder (EAM Think Tank, Germany)
- Joseph W. Yoder (The Refactory, USA)
- Uwe Zdun (TU Wien, Austria)

Produktlinien im Kontext: Technologie, Prozesse, Business und Organisation (PIK2010)

Andreas Birk[1], Klaus Schmid[2], Markus Völter[3]

[1] Software.Process.Management
andreas.birk@swpm.de
[2] Universität Hildesheim
schmid@sse.uni-hildesheim.de
[3] voelter - ingenieurbüro für softwaretechnologie/itemis
voelter@acm.org

Produktlinien sind heute in vielen Bereichen der Software-Industrie vertreten, von eingebetteten Systemen bis zu betrieblichen Informationssystemen. Sie ermöglichen höhere Produktivität, steigern die Qualität und verbessern die strategischen Positionen der Unternehmen, u.a. aufgrund eines hohen Grades an Wiederverwendung und Standardisierung.

Dennoch bergen Produktlinien für viele Unternehmen noch bedeutende Herausforderungen und Risiken. Die Gründe liegen teilweise im technischen Bereich. So sind viele Produktlinien-Technologien für den breiten Einsatz in der Praxis noch nicht genügend ausgereift und miteinander integriert. Die wohl größten Herausforderungen stellen sich in den Wechselwirkungen zwischen den technischen Verfahren mit den Prozessen sowie dem organisatorischen und geschäftlichen Kontext der Produktlinienentwicklung.—Wie müssen die technologischen Ansätze auf diese Wechselwirkungen ausgerichtet sein? Welche Potenziale bieten neue technologische Entwicklungen in unterschiedlichen Einsatzfeldern?

Der Workshop „Produktlinien im Kontext“ will aktuelle Erfahrungen mit Produktlinien beleuchten und den Dialog zwischen Praxis und anwendungsorientierter Forschung fördern. Im Mittelpunkt steht das Wechselspiel zwischen technischen Fragestellungen und den geschäftlichen, organisatorischen und Prozessaspekten. Daneben sollen auch neue technologische Entwicklungen vorgestellt und diskutiert werden.

Innovative Systeme zur Unterstützung der zivilen Sicherheit: Architekturen und Gestaltungskonzepte (Public Safety)

Rainer Koch[1], Margarete Donovang-Kuhlisch[2], Benedikt Birkhäuser[1]

[1]Universität Paderborn / C.I.K.
r.koch@cik.uni-paderborn.de, b.birkhaeuser@cik.uni-paderborn.de
[2]IBM Deutschland GmbH
mdk@de.ibm.com

Zunehmend rücken Fragen der zivilen Sicherheit in den Fokus der Aufmerksamkeit. Entsprechend sind in den letzten Jahren verstärkt Forschungsanstrengungen entstanden, um IT-Systeme zur Unterstützung von Akteuren in der Domäne zu entwickeln.

Ziel des Workshops ist es, die maßgeblichen Systementwickler aus diesem Bereich zusammenzubringen um Ansätze auszutauschen und zu diskutieren. Thematischer Inhalt des Workshops sollen dazu insbesondere zwei wiederkehrende Herausforderungen sein: Die Einbettung der Lösungen in domänenspezifische Randbedingungen und der Aufbau flexibler Lösungen.

2. Workshop für Requirements Engineering und Business Process Management (REBPM 2010)

Daniel Lübke[1], Kurt Schneider[2], Jörg Dörr[3], Sebastian Adam[3], Leif Singer[2]

[1]innoQ Schweiz GmbH
daniel.luebke@innoq.com
[2]Leibniz Universität Hannover, FG Software Engineering
{kurt.schneider, leif.singer}@inf.uni-hannover.de
[3]Fraunhofer IESE
{joerg.doerr, sebastian.adam}@iese.fraunhofer.de

SOA ist ein aufstrebender Architekturstil für große Softwaresysteme, aber auch für ganze Unternehmen und ihre Anwendungslandschaften. SOA ist daher nicht auf die IT-Abteilung und die Softwareentwicklung beschränkt, sondern wird Unternehmen in ihrer Gesamtheit betreffen.

Requirements Engineering als Schnittstellendisziplin zwischen "Kunde" und "Entwickler" muss sich in diesem Kontext neu definieren, da klare Auftraggeber-Auftragnehmer-Situationen in den Hintergrund treten und die Unternehmen integriert von der Strategie bis zum Betrieb der IT ganzheitlich gestaltet werden müssen.

In diesem Kontext lassen sich fachliche Lösungen (optimierte Geschäftsprozesse) nur schwer von technischen Lösungen (Anwendungssystemen) trennen, da beide eng verknüpft ("aligned") sein müssen, um die SOA-Vision des flexiblen Unternehmens tatsächlich umsetzen zu können. Dies beeinflusst insbesondere auch die Anforderungserhebungsphase, die nun viel mehr auf die ganzheitliche Umsetzung der Geschäftsziele und -abläufe unter Beachtung der Fähigkeiten der IT eingehen muss.

Als Weiterführung des REBPM 2009 soll auch REBPM 2010 wieder interessante Fragen im Bereich von Geschäftsprozessmodellierung und Requirements Engineering beleuchten. Hierbei sollen interessante Lösungen aus der Praxis und der Forschung präsentiert, sowie offene Fragestellungen identifiziert und diskutiert werden.

Unter dem Titel "Requirements Engineering und Business Process Management - Konvergenz, Synonym oder doch so wie gehabt?" laden wir daher wieder zu einem interdisziplinären Workshop ein, der diesmal auf der SE 2010 in Paderborn stattfinden wird.

Workshop on Social Software Engineering (SSE2010)

Wolfgang Reinhardt[1], Martin Ebner[2], Imed Hammouda[3],
Hans-Jörg Happel[4], Walid Maalej[5]

[1]Universität Paderborn
wolle@uni-paderborn.de
[2]Technische Universität Graz
martin.ebner@tugraz.at
[3]Tampere University of Technology
imed.hammouda@tut.fi
[4]FZI Forschungszentrum Informatik Karlsruhe
happel@fzi.de
[5]Technische Universität München
maalejw@in.tum.de

Software wird von Menschen, mit Menschen und für Menschen hergestellt. Diese Menschen arbeiten in unterschiedlichen Umgebungen, haben verschiedene Hintergründe und agieren unter vielfältigen Einflüssen. Daher ist es von besonderer Bedeutung zu verstehen, wie die menschlichen und sozialen Aspekte des Software Engineerings sowohl Methoden und Werkzeuge als auch die erstellten Softwaresysteme selbst beeinflussen. Social Software Engineering (SSE) beschäftigt sich mit der Softwareentwicklung in den unterschiedlichsten Wissensgebieten, mit sich entwickelnden Zielen, häufigen Änderungen und der regelmäßigen Einbindung des Endanwenders. In den letzten Jahren hat die Forschung im Bereich des Software Engineering gezeigt, dass neben fortschrittlichen Werkzeugen und Methoden vor allem effektive Kommunikation und Zusammenarbeit, Wissensaustausch und interdisziplinäres Verständnis für kognitive Prozesse wichtig für den Erfolg von Softwareprojekten ist.

Obwohl sowohl die Entwicklung sozialer Software als auch die sozialen Faktoren im Software Engineering entsprechende Aufmerksamkeit in der Community finden, sind wir der Meinung, dass beide zu einem neuen Software Engineering Paradigma zusammenfließen, das besonderer Betrachtung bedarf.

Software-Qualitätsmodellierung und -bewertung (SQMB)

Stefan Wagner[1], Manfred Broy[1], Florian Deißenböck[1], Jürgen Münch[2], Peter Liggesmeyer[2]

[1]Technische Universität München
{wagnerst, broy, deissenb}@in.tum.de
[2]Fraunhofer-Institut für Experimentelles Software Engineering, Kaiserslautern
{juergen.muench, peter.liggesmeyer}@iese.fraunhofer.de

Software-Qualität ist ein entscheidender Faktor für den Erfolg eines softwareintensiven Systems. Die Beherrschung der Qualität stellt aber immer noch eine große Herausforderung für Praxis und Forschung dar. Problematisch ist auch die Vielschichtigkeit und Komplexität von Qualität, die zu einer Vielzahl von nicht integrierten Insellösungen geführt hat. Eine umfassende Behandlung von Qualität wird typischerweise durch Qualitätsmodelle und darauf aufbauenden Bewertungen erwartet. Leider können Standard-Qualitätsmodelle, wie die ISO 9126 und ISO 25000, in der Praxis nur schwer angewandt werden, was zu einer Vielzahl von individuellen Qualitätsmodellen geführt hat.

Dieser Workshop hat das Ziel, Erfahrungen mit Qualitatsmodellierung und -bewertung zu sammeln, in eine gemeinsame Landkarte einzuordnen und gemeinsam neue Forschungsrichtungen zu entwickeln.

GI-Edition Lecture Notes in Informatics

P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.

P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.

P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Language to Information Systems, NLDB'2001.

P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensorgestützte Chirurgie; Workshop des SFB 414.

P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.

P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.

P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.

P-8 Reinhard Keil-Slawik, Johannes Magenheim (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.

P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.

P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.

P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.

P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.

P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.

P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.

P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.

P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Busi-ness 2002.

P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.

P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.

P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3.Okt. 2002 in Dortmund.

P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3.Okt. 2002 in Dortmund (Ergänzungsband).

P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.

P-22 Sigrid Schubert, Johannes Magenheim, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.

P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit

P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen

P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003

P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web

P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin

P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Erfahrungen und Visionen

P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems

P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications

P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometrics and Electronic Signatures

P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003

P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft

P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)

P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)

P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenberg (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik

P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik

P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003

P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003

P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004

P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing

P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste

P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistance, Scalability, Transactions – Database Mechanisms for Mobile Applications

P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning. E-Services

P-45 Bernhard Rumpe, Wofgang Hesse (Hrsg.): Modellierung 2004

P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment

P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society

P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications

P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven

P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik

P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004

P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration

P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Adhoc-Netze Medienintegration

P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government

P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen

P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality

P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Ranneberg, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments

P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für inforrmatische Bildung

P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen

P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit

P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and ist Applications

P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005

P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web

P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik

P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)

P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)

P-69 Robert Hirschfeld, Ryszard Kowalcyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005

P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)

P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005

P-72 Klaus P. Jantke, Klaus-Peter Fähnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment

P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): “Heute schon das Morgen sehen“

P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology

P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture

P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz

P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006

P-78 K.-O. Wenkel, P. Wagner, M. Morgenstern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel

P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006

P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce

P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS´06

P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006

P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics

P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications

P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems

P-86 Robert Krimmer (Ed.): Electronic Voting 2006

P-87 Max Mühlhäuser, Guido Rößling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik

P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006

P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur

P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006

P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006

P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1

P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2

P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen

P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies

P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices

P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik

P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik

P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten

P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments

P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)

P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme

P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007

P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops

P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications

P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures

P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1

P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2

P-111 Christian Eibl, Johannes Magenheim, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik

P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis

P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)

P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics

P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007

P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007

P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and manegement The 4th International Conference on Grid Service Engineering and Management GSEM 2007

P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007

P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications

P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges

P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik

P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)

P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Speciel Interest Groups PARS and PARVA

P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT: Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung

P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)

P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008

P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany

P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems

P-130 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
1. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung

P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik

P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 1

P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 2

P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik – Aktuelle Forschungsergebnisse

P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008

P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures

P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008

P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwick-lung und -wartung im Fachbereich Wirtschaftsinformatik

P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics

P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informations-systeme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management

P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung

P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)

P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management

P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)

P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009

P-148 Christian Erfurth, Gerald Eichler, Volkmar Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I²CS 2009

P-149 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
2. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband

P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software

P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009

P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik

P-154 Stefan Fischer, Erik Maehle Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben

P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«

P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics 2009

P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture Reloaded – Informationsgestützte Landwirtschaft

P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010

The titles can be purchased at:

Köllen Druck + Verlag GmbH
Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
Fax: +49 (0)228/9898222
E-Mail: druckverlag@koellen.de