TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

Informatik-Bericht Nr. 2009-05

LaZuSo 2009

2. März 2009

1. Workshop für langlebige und zukunftsfähige Softwaresysteme 2009 – SE Konferenz 2009 Kaiserslautern –

Andreas Rausch (TU Clausthal)
Ursula Goltz (TU Braunschweig)
Gregor Engels (Universität Paderborn)
Michael Goedicke (Universität Duisburg-Essen)
Ralf Reussner (Universität Karlsruhe (TH))

Institut für Programmierung und Reaktive Systeme

1 Einleitung

Eine weitverbreitete Annahme ist, dass Software nicht altert, da Software als immaterielles Gut keinerlei Verschleißerscheinungen unterliegt. Diese Annahme ist aber falsch - Software altert: Die Umgebung in der Software eingesetzt wird verändert sich kontinuierlich. Zum Beispiel verändern sich die Organisationen und deren Prozesse, in deren Rahmen die betrachtete Software verwendet wird. Oder aber auch die unterliegende Hardware und Infrastruktur verändert sich. Wird die Software nicht ebenfalls kontinuierlich weiterentwickelt, so altert sie relativ zu ihrer Umgebung, die sich weiter entwickelt.

Deshalb müssen insbesondere große Softwaresysteme ebenfalls kontinuierlich weiter entwickelt werden, damit sie nicht "altersschwach" werden. Nur so können die Unternehmen die erheblichen Investitionen, die in die Softwaresysteme getätigt wurden, schützen. Darüber hinaus kann meist eine vollständige Neuentwicklung der betroffenen Softwaresysteme nicht ernsthaft in Betracht gezogen werden, auf Grund der hohen Kosten und Risiken sowie des fehlenden Knowhow und der damit verbundenen Komplexität. Somit ist die einzig verbleibende Möglichkeit, die bestehenden Softwaresysteme kontinuierlich weiter zu entwickeln.

Ziel dieses Workshops war es Verfahren und Methoden zu diskutieren, um langlebige und auch in der Zukunft tragfähige (manageable) Softwaresysteme bauen und betreiben zu können. Oder anders gesagt: Wie vermeiden wir heute die Legacy-Systeme von morgen? Hierzu erscheint ein ganzheitliches, architekturzentriertes Management von Softwaresystemen notwendig. Dieses umfasst nicht nur die Entwicklung sondern auch den Betrieb und die Evolution dieser Softwaresysteme. Darüber hinaus müssen hierbei alle Teilgebiete des Software-Engineering angefangen von den technischen, wie zum Beispiel Requirements-Engineering, Architekturentwurf, Design und Implementierung sowie Qualitätssicherung mit den unterschiedlichen querschnittlichen Disziplinen wie zum Beispiel Modellierung, Spezifikation, Verifikation, etc., wie auch die nicht-technischen wie zum Beispiel Projekt- und Organisationsmanagement, IT-Systemmanagement und ökonomische und wirtschaftliche Aspekte, betrachtet werden. Dabei spielen stets die Produktqualitätseigenschaften, wie Adaptivität, Performanz, Zuverlässigkeit, wie auch die Prozessqualitätseigenschaften, wie Effizienz, Kosten, Zeit, eine wesentliche Rolle.

Im Rahmen dieses Workshops stand die Frage im Zentrum, welchen Beitrag die angesprochenen Teildisziplinen und Aspekte des Software-Engineering jeweils aus ihrer unterschiedlichen Perspektive für das anvisierte Ziel, langlebige und zukunftsfähige Softwaresysteme entwickeln und betreiben zu können, beitragen können.

2 Vorträge

- Barbara Paech: Begrüßung und Entstehung des Workshops durch den Fachbereich Softwaretechnik der GI
- Ursula Goltz und Ralf Reussner: Einführung in das Thema: Workshop für langlebige und zukunftsfähige Softwaresysteme
- Kurt Schneider: Software Engineering Rationale: Wissen über Software erheben und erhalten
- Michael Goedicke: Langlebigkeit von Systemen und Systembestandteilen durch Variantenmanagement, Service-Orientierung und Co-Evolution von Anforderungen und Architektur
- Barbara Paech: Business-IT-Alignment durch Requirements Engineering
- Sabine Buckl: Enterprise Architecture Management Patterns für zukunftsfähige Anwendungslandschaften
- Lutz Prechelt: Sustainable SW: Learning from Open Source Software Development
- Andy Schürr: Modellbasierte Architekturevolution verteilter Software-Systeme
- Bernhard Rumpe: Modellbasierte Evolution als Schlüssel nachhaltig anpassbarer Software Systeme
- Maritta Heisel: Muster als Mittel zur Entwicklung langlebiger Software
- Jens Grabowski: Anmerkungen zur Langlebigkeit von Testartefakten
- Arnd Poetzsch-Heffter: Modeling Evolving Systems: The HATS Approach
- Wolf Zimmermann: Robuste und wiederverwandbare Softwaresysteme durch Komponentenprotokollprüfung
- Jürgen Ebert: Softwareevolution durch Adaptivität und Adaptierbarkeit
- Guido Gryczan: Von Anwendungssoftware zu Anwendungslandschaften
- Lutz Prechelt: The Empirical Evaluation Service Provider
- Abschlussdiskussion

3 Zusammenfassung der Ergebnisse

In diesem Abschnitt werden die Ergebnisse des Workshops durch Abstracts der einzelnen Vorträge (in alphabetischer Reihenfolge der Autoren) dargestellt.

Enterprise Architecture Management Patterns für zukunftsfähige Anwendungslandschaften

Sabine Buckl, Florian Matthes

Die Langlebigkeit und steigende Anzahl von Softwaresystemen in Unternehmen führen zu immer komplexeren Anwendungslandschaften, die von Personen mit sehr unterschiedlichen Interessen und Erfahrungshintergrund konzipiert, erstellt, modifiziert, betrieben, genutzt und finanziert werden. Die gesteuerte Evolution der Anwendungslandschaft ist eine der Kernaufgaben des Enterprise Architecture (EA) Managements, welches eine ganzheitliche Betrachtung des Unternehmens inklusive fachlicher, technischer und wirtschaftlicher Aspekte ermöglicht.

Dabei unterstützt das EA Management die Kommunikation zwischen den beteiligten Personen durch eine einheitliche Terminologie und zielgruppen-spezifische Visualisierungen, die speziell für ganzheitliche und strategische Betrachtungen geeignet sind. Der Kontext eines Softwaresystems - die unterstützten Geschäftsprozesse, die genutzte Hardware und Infrastruktur etc. - ist typischerweise sehr komplex und unterliegt ständigen Veränderungen. Die Entwicklung von langlebigen und zukunftsfähigen Softwaresystemen verlangt demnach nach Lösungsbausteinen, die eine Betrachtung des Softwaresystems in seinem fachlichen, wirtschaftlichen und technischen Kontext erlauben.

Der an der TU München im Aufbau befindliche EA Management Pattern Catalog stellt wiederverwendbare Lösungsbausteine basierend auf Best-Practices aus Industrie und Forschung für eine Vielzahl wiederkehrender EA Problemstellungen bereit. Diese Lösungsbausteine sind analog zu Patterns aus dem Software Engineering dokumentiert und enthalten neben einem Problem-, Lösungs- und Konsequenzenabschnitt auch detaillierte Informationen über ihren Einsatzkontext. Der Ansatz verwendet drei Arten von EA Management Pattern. Sogenannte Methodology Pattern sind in Form einer Anleitung formuliert und beschreiben Management-Methoden, um typische EA Fragestellungen zu beantworten. Ein Information Model Pattern definiert ein konzeptuelles Datenmodell für den relevanten Teil der EA, z.B. dem Kontext eines Softwaresystems, und spezifiziert damit die im Unternehmen zu pflegenden Model-

linformationen. Ein Viewpoint Pattern zeigt, wie die Daten des konzeptuellen Datenmodells in Visualisierungen aufbereitet werden können. Am Lehrstuhl entwickelte Werkezeuge (System Cartography Tool, Hybrid Wiki) verfolgen einen metamodellbasierten Ansatz, um das musterbasierte Enterprise Architecture Management zu unterstützen.

Erst im Kontext eines EA Managements kann ein zukunftsfähiges und nachhaltiges Management betrieblicher Softwaresysteme sichergestellt werden. Die dabei verfolgte ganzheitliche und modellbasierte Betrachtung von Systemen in Ihrem betrieblichen Kontext ermöglicht eine gesteuerte Evolution der Softwaresysteme über die einzelnen Lebenszyklusphasen Planung, Entwicklung, Betrieb und Wartung.

Softwareevolution durch Adaptivität und Adaptierbarkeit

Jürgen Ebert

Die Erfordernisse an die Anpassung von Software an veränderte Anforderungen sind teilweise vorhersehbar, teilweise aber auch überraschend. Adaptive Systeme sind (in gewissem Rahmen) zur Selbstanpassung geeignet sind und können den Anpassungsprozess selbst vollziehen. Adaptierbare Systeme hingegen gestatten (nur) die leichte Anpassung durch externe Maßnahmen.

In diesem Vortrag wird die These vertreten, dass Langlebigkeit von Software durch eine adäquate Balance zwischen Adaptivität und Adaptierbarkeit unterstützt werden soll und dass modell-basierte Ansätze gute Voraussetzungen zur Entwicklung solcher Systeme bieten.

Bei modell-basierter Entwicklung dienen Modelle als Basis der Softwareentwicklung. In diesem Fall wird aus passenden Modellen durch Transformationen und Generierungsprozesse das Zielsystem erzeugt, in dem die Modelle dann in der Struktur der Software aufgehen (implizite Modelle). Alternativ können Modelle auch zur Laufzeit zugänglich bleiben (explizite Modelle), was zu generischen Systemen führt. In diesem Falle werden die Modelle zur Laufzeit interpretiert und bestimmen dadurch explizit das Verhalten des Systems.

Modelle in diesem Sinne können grundsätzlich alle Beschreibungen von Aspekten eines Softwaresystems sein. Solange eine Beschreibungssprache für einen Aspekt definiert ist (etwa durch ein Metamodell), kann meist der abstrakte Syntaxgraph (eine Instanz des Metamodells) einer konkreten Beschreibung als Objekt der Interpretation dienen.

Beispiele für Modelle, deren Interpretation zur Laufzeit Sinn machen könnte, sind beispielsweise geometrische Modelle in mobilen Systemen, Ablaufbeschreibungen in prozess-leittechnischen Anwendungen oder Richtlinien (policies) im Datenschutzbereich.

Die Adaption von Modellen zur Laufzeit erfordert zuerst ein Monitoring des Softwareverhaltens in Bezug auf das Modell sowie adäquate Metriken, die es erlauben, die Monitoring-Daten zu beurteilen. Regelsysteme erlauben dann, abhängig von den Metrikdaten Modelltransformationen durchzuführen. Ist die Konsistenzerhaltung für die eine Regel bewiesen, so ist eine korrekte Laufzeittransformation (und damit eine Adaption des Verhaltens) möglich. Auf diesem Wege ist eine weitgehende Adaptivität durch Modelle zur Laufzeit erreichbar.

Metrikdaten können allerdings auch zur Erkennung von Situationen führen, die eine weitere Adaption nicht mehr als sinnvoll erscheinen lassen, weil eine Restrukturierung des Systems (z.B. der expliziten Modelle) von außen erforderlich ist. In diesem Falle ist Adaptierung durch Reengineering erforderlich. Reengineering kann dann so weit gehen, dass auch die Modellsprache angepasst werden muss. Durch entsprechendes Metamodell-Engineering wird diese Adaptierbarkeit erreicht.

Der hier skizzierte Ansatz führt zu einer Reihe von Forschungsfragen, wie beispielsweise: Welche Modelle sind für welche Selbstadaptionen am Besten geeignet? Welche Metriken zu einem Modell korrelieren mit der Notwendigkeit zur Selbstanpassung? Wie leitet man zielgerichtet Anpassungsregeln her? Wie erkennt man, dass Adaption durch Reengineering erforderlich ist? Welcher Transformationsmechanismus ist adäquat und wie verifiziert man die Korrektheit der Regeln?

In diesem Vortrag wird vorgeschlagen, diesen Ansatz in mit graphentechnologischen Mitteln zu behandeln, die eine explizite Repräsentation von Metamodellen und Modellen in einem formal zugänglichen und effizient implementierbaren Kontext erlauben.

Anmerkungen zur Langlebigkeit von Testartefakten

Jens Grabowski

Während Probleme aufgrund der Langlebigkeit von produktiven Softwaresystemen seit den Diskussionen um die Jahr 2000-Umstellung im Bewusstsein der Softwarehersteller sind, gibt es bisher kaum Untersuchungen zur Langlebigkeit von anderen Artefakten, die vor oder während der Softwareentwicklung angefertigt werden. In diesem Vortrag wird die Langlebigkeit von Testartefakten basierend auf ausgewählten Beispielen diskutiert.

Testen ist die wichtigste Methode zur Qualitätssicherung von Softwaresystemen. Testaktivitäten wie die Planung, Implementierung, Durchführung, Ausführung und Auswertung von Tests sind daher ein integraler Bestandteil aller Phasen im Software-Lebenszyklus. Im Rahmen dieser Testaktivitäten werden verschiedene Testartefakte auf unterschiedlichen Abstraktionsebenen erstellt. Akzeptanz- und Systemtests basieren meist auf Anwendungsfällen oder Geschäftsprozessen, benötigen Benutzerinteraktionen und sind daher häufig sehr abstrakt, d.h. ohne konkrete Eingabedaten,

spezifiziert. Demgegenüber versucht man Komponenten- und Regressionstests üblicherweise auf einer niedrigen Abstraktionsebene, d.h. als ausführbare Programme, zu realisieren.

Es gibt Beispiele für Akzeptanz- und Systemtests, die mehrere Softwaresysteme überleben und immer noch aktuell angewendet werden. Häufig handelt es sich hier um Testfälle für Geschäftsmodelle, die auch ohne oder mit wenig Softwareunterstützung funktionieren. Als Beispiel sei hier Lotto genannt, dessen Geschäftsmodell und die zugehörigen Anwendungsfälle ("Lottoschein abgeben", "Gewinn abholen", usw.) sich über mehrere Jahrzehnte kaum verändert haben. Ein hohes Abstraktionsniveau ist jedoch keine notwendige Voraussetzung für Langlebigkeit. Die Akzeptanz- und Systemtests für ISDN-Endgeräte wurden z.B. vor mehr als 20 Jahren mit der semiformalen "Tree and Tabular Combined Notation" (TTCN) spezifiziert und standardisiert. Obwohl die ISDN-Technologie nicht mehr weiterentwickelt wird, werden immer noch ISDN-Endgeräte verkauft, die dann mittels Gateway-Funktionen mit anderen Kommunikationstechnologien verbunden werden. Sowohl die Gateway-Funktionen wie auch die neuen ISDN-Endgeräte müssen vor einer Zulassung im Telefonnetz auf ihre Konformität zu den ISDN-Standards getestet werden. Basis für diese Tests bilden die alten, vor über 20 Jahre alten standardisierten, ISDN-Tests. Schwierigkeiten bei der Durchführung dieser Tests entstehen dadurch, dass viele ISDN- und ISDN-Test-Spezialisten inzwischen im wohlverdienten Ruhestand sind, dass kaum noch ISDN-Testwerkzeuge angeboten werden und das sich die Migration von TTCN zur neuesten Testtechnologie, der "Testing and Test Control Notation" (TTCN-3), als sehr aufwendig herausgestellt hat.

Die Langlebigkeit und Stabilität von Komponententests lässt sich sehr schwer untersuchen, da es sehr schwierig ist "alte" Komponententests zu finden. In der Open-Source-Bibliothek des Eclipse-Projekts finden sich zumindest Beispiele für sehr stabile Komponententests. Solche Testfälle wurden mit dem ersten Release der Eclipse-Plattform veröffentlicht. Sie wurde seither erweitert und gewartet, aber die Struktur wurde nicht verändert. Komponententests, die die zu testende Komponente überlebt haben konnten wir bisher nicht finden.

Die obigen Beispiele zeigen, dass nicht nur produktive Softwaresysteme, sondern auch andere Software-Artefakte, wie z.B. Testspezifikationen, sehr langlebig sein können. Dieses sollte bei der Software-Entwicklung mit berücksichtigt werden, in dem alle Software-Artefakte einer konsequenten Qualitätssicherung unterzogen werden. Als Beitrag hierfür erforscht und entwickelt die Forschungsgruppe "Softwaretechnik für verteilte Systeme" am Institut für Informatik der Georg-August-Universität Göttingen Techniken für die systematische Qualitätsbewertung und -verbesserung von Testspezifikationen.

Von Anwendungssoftware zu Anwendungslandschaften

Guido Gryczan

Die Forschung in der Softwaretechnik beschäftigt sich vorrangig mit dem systematischen Einsatz von Programmiersprachen und Methoden und Techniken zur Entwicklung einzelner (Anwendungs-) Softwaresysteme. Auch wenn die Durchdringung der industriellen Praxis mit diesen Forschungsergebnissen noch nicht wirklich befriedigend ist, stellen wir fest, dass es keine technischen Gründe mehr gibt, mit denen das Scheitern von Entwicklungsprojekten argumentiert werden kann.

Etwas anders schauen wir auf die Lage, wenn es um die Weiterentwicklung von Anwendungssoftware in sog. Anwendungslandschaften geht.

Offenbar müssen aber schon für die IT-Unterstützung scheinbar einfacher Geschäftsprozesse - etwa: Abfrage eines Kontostandes über das Internet - mehrere Softwaresysteme aus verschiedenen Sprach- und Technologiegenerationen zusammenwirken. Das Spektrum in einer Anwendung reicht häufig von objektorientierten Programmiersprachen zur Programmierung der Benutzungsoberfläche bis zu Host-basierten Applikationen in denen Teile der Anwendungslogik programmiert sind.

Wir sprechen in diesem Zusammenhang von Anwendungslandschaften. Eine wesentliche Herausforderung an Softwaretechniker und Software-Architekten besteht darin, Techniken und Herangehensweisen zur Analyse und Weiterentwicklung solcher Anwendungslandschaften zu entwickeln. Diese Aufgabe ist herausfordernd für eine universitäre Informatik, da sie nach unserer Erfahrung ohne einen vertieften Blick auf die industrielle Praxis nicht zu bewältigen ist.

Im Vortrag wird beispielhaft erläutert, wie das Thema “Migration von Anwendungslandschaften” am Dept. Informatik der Univ. Hamburg angegangen wird. Dazu werden Beispiele aus von uns begleiteten Migrationsprojekten präsentiert.

Im Kern des Vortrags beschäftigen wir uns mit der Frage, wie wir in den von uns betrachteten Migrationsprojekten mit unvollständigen Modellen von Anwendungslandschaften umgegangen sind. Aufgrund der praktischen Natur der Migrationsprojekte hat es sich (für uns) als unmöglich erwiesen, vollständige Modelle der Anwendungslandschaften zu erstellen. Wir zeigen deshalb auf, wie mit der exemplarischen Geschäftsprozeßmodellierung maximierte Handlungssicherheit bei unvollständig modellierter Ausgangslage erreicht werden kann.

Abschließend diskutieren wir die Frage, ob es sinnvoll ist, Vollständigkeit bei der Modellierung zu migrierender Anwendungslandschaften anzustreben.

Muster als Mittel zur Entwicklung langlebiger Software

Maritta Heisel

Muster sind ein etabliertes Mittel der Softwaretechnik, das die Entwicklung von Software dadurch erleichtert, dass im Laufe der Zeit angesammeltes Wissen wiederverwendet werden kann. Gegenwärtig werden hauptsächlich Muster verwendet, die der Strukturierung von Software-Lösungen dienen (z.B. Architektur- und Entwurfsmuster). Es gibt jedoch auch Muster, die der Einordnung von Software-Entwicklungsproblemen in bekannte Klassen (z.B. Regelungs- oder Transformationprobleme) dienen (problem frames).

Die möglichst durchgängige Verwendung verschiedener Musterarten im Softwareentwicklungsprozess kann wesentlich zur Langlebigkeit von Software beitragen. Dies wird dadurch erreicht, dass sowohl Problem- als auch Lösungsbeschreibungen Instanzen von Mustern sind, demnach also durch Wiederverwendung gewonnen werden. Problem- und Lösungsmuster sollten durch heuristische Zuordnungen zueinander in Beziehung gesetzt werden. Dies trägt der Tatsache Rechnung, dass für ähnliche Probleme auch ähnliche Lösungsansätze gewählt werden können. Designalternativen können auf verschiedenen Wegen gewonnen werden, z.B. durch verschiedene Problemzerlegungen, Verwendung unterschiedlicher Zuordnungen von Problem- zu Lösungsmustern, oder durch Verwendung unterschiedlicher Komponenten.

Auch die für die Langlebigkeit unerlässliche Evolution von Software wird durch diesen Ansatz unterstützt. Veränderte Anforderungen können zur Verwendung weiterer Muster bzw. zum Austausch eines Musters gegen ein anderes führen. Durch die Dokumentation der zur Verfügung stehenden Muster sowie ihrer Beziehungen ist dies in systematischer Weise möglich.

Insgesamt lässt sich feststellen, dass die Verwendung von Mustern in möglichst vielen Phasen der Softwareentwicklung und das In-Beziehung-Setzen der verschiedenen Musterarten ein vielversprechender Ansatz zur Entwicklung und Erhaltung langlebiger Software ist.

Business-IT-Alignment durch Requirements Engineering

Barbara Paech

Aus Sicht des Requirements Engineering ist eine Software dann langlebig und zukunftsfähig, wenn sie

1. die *Geschäftsanforderungen* erfüllt (Alignment)

2. sich *kontinuierlich* (auch zur Betriebszeit) an Änderungen der Geschäftsanforderungen anpassen lässt und

3. die *Qualität des Alignment* zur Laufzeit überprüft wird und so einerseits ein Feedback zu durchgeführten Anpassungen möglich ist und andererseits Lücken für weitere Anpassungen aufgezeigt werden

Die Literatur (insbes. der Wirtschaftinformatik) beschäftigt sich schon lange mit dem Business-IT-Alignment. Typischerweise werden dabei 4 Bereiche miteinander abgeglichen

- Geschäftsstrategie
- Geschäftsorganisation: Unternehmensmodell und Geschäftsprozessarchitektur
- IT-Strategie
- IT-Organisation: IT-Architektur und -Prozesse

Konkrete Vorgehensmodelle dafür fehlen aber, auch in den gängigen Geschäftsprozessmodellansätzen wie ARIS, da entweder die strategische Seite oder der Übergang zu den IT-Anforderungen nicht berücksichtigt wird. Dies wird insbesondere auch deutlich im Umfeld der neueren Forschung im Bereich Geschäftsprozessmodellierung mit BPMN und service-orientierten Architekturen mit SOA. Insbesondere ist das Alignment auch abhängig vom Geschäftskontext: geht es um das Alignment zwischen einem Softwareproduzierenden Geschäftskontext und dem Softwareprodukt oder um das Alignment zwischen einem beliebigen Geschäftskontext und der IT-Infrastruktur dafür?

Es wird deshalb vorgeschlagen, Methoden und Werkzeuge des Business-IT-Alignments zu erarbeiten, die insbesondere eine kontinuierliche Überprüfung der Qualität (also der Zukunftsfähigkeit und der Langlebigkeit der Software in Bezug auf den Geschäftskontext) ermöglichen. Dazu sind die folgendenen Forschungsfragen zu beantworten:

- Beschreibungstechniken: wie beschreibt man IT/Produkt-Strategie und wie beschreibt man Unternehmensmodell und Prozessarchitektur als Basis für den Abgleich mit den IT/Produkt-Anforderungen bzw. mit der IT/Produktarchitektur?
- Abhängigkeiten: Wie beeinflussen sich Business- und IT/Produkt-Architektur? Wo schränkt eines das andere ein? Wo sind Synergien?
- Vorgehensmodell: Wie ermöglicht man einen kontinuierlichen Abgleich?

- Qualität: Wie kann man Qualität von Business und IT/Produkt gleichzeitig bewerten und optimieren? Welche Daten muss man erheben, um die Qualität von Business und IT/Produkt messen zu können, und wie kann man diese bei der Weiterentwicklung nutzen?
- Werkzeuge: Wie kann man dieses Vorgehen durchgängig durch Werkzeuge unterstützen?

Modeling Evolving Systems: The HATS Approach

Arnd Poetzsch-Heffter

Modeling approaches to evolving systems have to deal with at least three fundamental questions:

1. What aspects of software systems can evolve?
2. Which models of software systems have to be taken into account for evolution?
3. How can the relation between a system before and after an evolution step be expressed?

Unfortunately, evolution is a quite diverse concept: It can be caused by changing functional or non-functional requirements of the system behavior; e. g. new features could be required or a higher degree of availability. It can be related to the implementation of the system, e. g. to adapt it to a new version of an operating system. In general, it can even happen at the installation level, e. g. if we want to migrate an information system to a new database management system. A further challenge for modeling evolving systems is the fact that these different aspects of a system or a system family are usually expressed by different, often only loosely related models. The HATS[1] approach to modeling evolving systems is based on two basic ideas:

1. Use a semantics-based behavioral modeling language as backbone for modeling
2. Use techniques from software family modeling to express variability and evolvability

Within the approach, a system model covers both behavioral requirements as well as implementation and installation features. Variability of a system, for example with respect to functional requirements or platform aspects is expressed by feature models

[1] HATS stands for Highly Adaptable & Trustworthy Software Using Formal Models. It is a European project with 11 partners (Integrated Project in FP7) with the goal to extend formal methods to highly adaptable and trustworthy software.

and feature specifications related to the behavioral model and implementation parts. The same feature-based modeling techniques are used to cope with evolution by treating evolution as variability over time.

In the talk, we sketch existing modeling techniques for different views of software systems, discuss reasons for evolution, in particular changing requirements, code maintenance, and technology improvements. Then, we describe the HATS approach as summarized above. Especially, we explain in which respect the behavioral modeling language is more abstract than high-level programming languages and discuss the integration with feature models and product derivation.

Langlebigkeit von Systemen und Systembestandteilen durch Variantenmanagement, Service-Orientierung und Co-Evolution von Anforderungen und Architektur

Klaus Pohl, Michael Goedicke, Kim Lauenroth

In Bezug auf die Langlebigkeit von Software-Systemen unterscheiden wir zwei unterschiedliche Perspektiven. Zum Einen betrachten wir die Langlebigkeit in der Systemanpassung: im Rahmen der Systemanpassung wird Langlebigkeit eines Systems realisiert, d.h. die Lebenszeit eines Systems wird durch Modifikation an geänderte Bedingungen verlängert. Zum Anderen betrachten wir Langlebigkeit in der Neudefinition von Systemen: im Rahmen der Systemneudefinition wird ein System aufgrund geänderter Bedingungen neu definiert. Einzelne Systembestandteile werden wiederverwendet, wodurch eine Langlebigkeit der Systembestandteile erreicht wird.

Im Folgenden erläutern wir, wie Langlebigkeit durch Variantenmanagement, Service-Orientierung und Co-Evolution von Anforderungen und Architektur erreicht werden kann.

Variantenmanagement hat das Ziel, die Varianten eines Systems explizit zu dokumentieren und zu verwalten. Das Variantenmanagement erlaubt die Anpassung eines Systems an vordefinierten Stellen in den Anforderungen, der Architektur, der Implementierung usw. Variantenmanagement verbessert die Langlebigkeit eines Systems, da durch die explizite Definition von Variabilität des Systems die Anpassbarkeit an zukünftige Entwicklungen erzielt werden kann. Des Weiteren kann durch Variabilität in einzelnen Systembestandteilen erreicht werden, dass die Systembestandteile angepasst werden können. Damit wird die Langlebigkeit einzelner Systembestandteile verbessert.

Service-Orientierung interpretiert ein System als eine Kombination von Services. Durch die lose Kopplung von Services wird eine hohe Flexibilität erzielt, d.h. eine nahezu beliebige Zusammenstellung von Services zu System. Service-Orientierung erlaubt den Austausch einzelner Services innerhalb eines Systems. Dadurch wird die Langlebigkeit verbessert, da das System durch Austausch von Services an geänderte

Bedingungen angepasst werden kann. Durch eine angemessene Granularität einzelner Services (im Sinne von "Business Funktionen") wird die Langlebigkeit einzelner Services verbessert, da diese in verschiedensten Systemen verwendet werden können.

Anforderungen und Architektur einen System beeinflussen sich gegenseitig und können daher nur gemeinsam entwickelt werden. Das Ziel der Co-Evolution von Anforderungen und Architektur besteht darin, Anforderungen und Architektur gemeinsam und konsistent zueinander in einem iterativen Prozess zu entwickelt. Die Co-Evolution von Anforderungen und Architektur erlaubt die konsistente Anpassung eines Systems, da Anforderungen und Architektur eines Systems gemeinsam und konsistent zueinander angepasst werden können. Des Weiteren können existierende Bestandteile leicht zu einem neuen System integriert werden, da die Anforderungen und die Architektur existierender Bestandteile im Rahmen der Co-Evolution des neuen Systems berücksichtigt werden können.

Im Rahmen unserer Forschungsvorhaben planen wir zu untersuchen, wie die Langlebigkeit von Systemen und Systembestandteilen durch die genannten Konzepte unterstützt und verbessert werden kann.

Sustainable SW: Learning from Open Source Software Development

Lutz Prechelt

One of the most prominent aspects of sustainable software development is the problem of design decay.

In principle, we already know how to develop software sustainably in this regard: Dave Parnas and others have explained why design structures decay in the course of many changes much like a building decays over time; Manny Lehman and others have quantified these effects for us. Since then, we had a vague idea that we need to invest effort to keep up the design structure over time. Then Mary Shaw and others explained the notion of architecture. Since then, we have a better idea of what it is that needs our continuous protection. Finally, Martin Fowler and others have provided us with a sort of basic instruction set for performing such archtectural upkeeping work: refactoring.

This means that whenever a software development turns out to be not sustainable, either the software organization is incompetent or short-sighted behavior has prevented the necessary activities from being performed. Therefore, sustainability of software is not a merely technical problem: it is a socio-technical one.

I believe that Open Source Software (OSS) projects represent a wonderful research opportunity for identifiying socio-technical practices that help (or hamper) sustainability: First, there are many examples of sizable OSS software, both sustainable and not-so-sustainable. Second, these examples (including the processes that led to their

creation) are public and non-confidential. Third, plenty of historical records exist about these developments, such as source code version archives, defect tracking databases, and mailing list archives. We just need to analyze them.

Furthermore, a number of typical properties of OSS processes are already known, we just need to find out how they influence sustainability: OSS projects work in a highly distributed manner, which often leads to highly modularized architectures. They use low-tech toolsets, but use them throughout. They employ continuous integration, make frequent releases, and have large numbers of beta testers. They produce documentation only when and where it is clearly needed. They have stable core teams, much larger surrounding teams, and fast career paths. All decision-making is performed by technical people only, using meritocratic distribution of authority, "do-ocracy" (power is with those people who invest the effort), and "lazy consensus building" and the level of irrationality that is possible in a project is bounded by the possibility of project forking.

I suggest a research project whose goal is identifying behavior patterns that help (or that hamper) sustainability by comparing successful to less successful project, mostly with respect to decision-making regarding architecture, to infrastructure (tools, components, methods, etc.), and to resource allocation. Such a research project would employ a combinatin of repository mining (source code version archive, bug tracker) and text-based qualitative methods (mailing list archive). It could use backward analysis, forward analysis, or both.

Backward analysis starts by identifying the OSS project's maintainability strengths/problems of today and then goes back and looks at the process how they were introduced and developed over time.

Forward analysis starts by identifying concerns and sustainability discussions of years ago and proceeds with assessing the later consequences of these discussions and decisions from year to year, until today.

The Empirical Evaluation Service Provider

Lutz Prechelt

Most Software Engineering (SE) research aims at usefulness (rather than just insight). However, usefulness can usually not be demonstrated by abstract arguments, but rather must be shown in practical application. Only there will the set of consequences of a would-be improvement become fully visible. Therefore, constructive SE research typically needs empirical evaluation of its products if it is to be convincing.

Good empirical work fulfills two quality criteria:

1. Credibility. This means the results reported are due to the factors investigated and not distorted by unknown factors or biased by partisan choice of question,

task, context or measurement method.

2. Relevance. This means the conditions of the study resemble conditions of practical interest rather than being artificially simple, isolated, or overly focussed.

Unfortunately, finding evaluation approaches that produce high credibility without sacrificing relevance or that produce high levels of both without exploding costs is very difficult.

The Software Engineering research group of Freie Universität Berlin offers all SE research groups that want to perform high-quality empirical evaluation but that would prefer outsourcing the related effort and expertise to perform such studies for them in a cooperative mode. Typically, we would design a study and carry it out in close cooperation with you; we then evaluate the results for you; both parties publish about the work together.

We have experience with the following research methods (which we have applied to these topics):

- Controlled experiments (type checking, design patterns, design pattern documentation, tour-shaped design documentation)
- Quasi-experiments (Personal Software Process, comparing 7 programming languages, comparing 3 web development platforms)
- Qualitative analysis of audio/video data (pair programming behavior patterns)
- Qualitative analysis of text data (innovation management, security-related aspects of development processes)
- Performance evaluation and benchmarking (design pattern detection in code, learning algorithms, compiler optimizations, music retrieval, knowledge management)
- Interviews, questionnaires (as an auxiliary technique in various cases)
- and also with case studies, simulation, and meta-analysis.

Modellbasierte Evolution als Schlüssel nachhaltig anpassbarer Software Systeme

Bernhard Rumpe, Steven Völkel

Software veraltet. Um Software Systeme, Software-Komponenten oder Wissen über Software langlebig zu erhalten, sind ähnliche Maßnahmen wie bei Maschinen oder

Gebäuden notwendig. Software Systeme wollen gepflegt, den Bedürfnissen angepasst, ggf. saniert und auch entmüllt werden.

Dabei ist das Problem nicht die Stabilität, sondern

- die Flexibilität mit einem gewandeltem Systemkontext umzugehen,
- die Migrationsfähigkeit auf neue Rechner-Hardware,
- die Anpassungsfähigkeit für neue Sensorik/Umgebungsinformation und
- die Anpassungsfähigkeit für geänderte/neue Geschäftsprozesse.

Dabei gibt es sehr wohl domänenspezifische Unterschiede, generell aber gilt, dass diese Probleme in ähnlicher Form sowohl bei eingebetteten Systemen verschiedener Coleur als auch bei langlebigen Business-Systemen auftreten.

Genereller Lösungsweg aus dieser Problematik besteht im Anheben des Abstraktionsgrades mit folgenden Hintergründen/Randbedingungen:

1. Die Abstraktion vom technischen Systemkontext führt zu stabileren, leichter migrierbaren Systemen. Abstraktion vom Systemkontext bedeutet unter anderem standardisierte Schnittstellen und Systemdienste (Betriebssystem, Network-Protokolle, Datenbanken).
2. Software und speziell ihre Strukturen sollte sich stark an Anforderungen orientieren. Idealerweise sollten Geschäftsprozesse und Anforderungen direkt abgebildet und nicht durch architektonische, optimierende oder sonstige Überlegungen zu sehr über die Codestruktur zerfasert werden. Dies führt zu deutlich wartbarerem und weiterentwickelbarem Code.
3. Abstraktion sollte auch kompaktere Programmierung/Modellierung erlauben. Damit ist trotz der Größe des Produkts eine schnellere Entwicklung und damit ein kleineres Projekt möglich. Die Entwicklungsartefakte sind kleiner und dieWeiterentwicklung damit flexibler.

Evolution steigert die Wiederverwendung und damit die Effektivität der Entwickler sowie typischerweise auch die Qualität des Ergebnisses, weil die modifizierte Form der Software auf die Erprobungsphasen vorangegangener Fassungen zurückgreifen kann.

Evolution von Code heißt heute die Planung von Umbauten im Kopf vorzunehmen und idealerweise durch eine Serie kleiner Refactorings umzusetzen. Leider integriert diese Form der Evolution des Systems in keinster Weise Artefakte der früheren Entwicklungsphasen.

Ideal wäre deshalb die Planung und Durchführung von Evolutionsschritten auf Basis von Modellen, die durch geeignete Generierungstechniken per Konstruktion

synchron zu Code sind und somit die Konsistenz zwischen dokumentierenden Modellen und Code sicherstellen. Aufgrund der den Modellen typischerweise inhärenten höheren Abstraktion und besseren Darstellungsform ist Refactoring auf Modellbasis besser planbar und effizienter durchführbar.

Refactoring auf Modellbasis bedeutet Evolution in kleinen, systematischen und überschaubaren Schritten. Systeme, die auf dieser Basis und mit der Idee der Refaktorisierbarkeit gebaut sind, sollten langlebig sein und gleichzeitig über ihre lange Lebensdauer hinweg Qualität und Zuverlässigkeit besitzen.

Voraussetzung für qualitätsgesicherte Modellevolution sind jedoch

- gute, intelligente Codegeneratoren,
- automatisierte Regressions-Tests und
- geeignete Sammlungen von Modelltransformationen.

Für einzelne UML-Notationen existieren bereits einige Refactoring-Techniken, teilweise als Übernahme von Refactorings aus dem Code (z.B. bei Klassendiagrammen), teilweise auch als mathematische Kalküle zur Modelltransformation (z.B. bei Statecharts, OCL).

Extreme Programming liefert die methodische Unterfütterung für die Programmierung, Eclipse, JUnit und Verwandte die Werkzeuge zur Durchführung und MDA die Methodik für die modellbasierte Softwareentwicklung. Sie nutzt Codegeneratoren und Transformationssprachen. Allerdings sind weder die Techniken ausgereift und ausgereizt, noch in breiter Masse in industrieller Stärke verfügbar.

So fehlen zum Beispiel gute Modularisierungstechniken für Modellierungssprachen ebenso wie für Transformationssprachen. Wenn wir Abstraktion, Konfiguration und Software- Planung auf Modellbasis einsetzen wollen, dann sind bessere Sprachen notwendig.

Modellbasiertes Software Engineering hat Potential für bessere Evolution, mehr Flexibilität des Produkts, stärkere Konfigurierbarkeit und Anpassbarkeit damit über den Lebenszyklus hinweg bessere Wartbarkeit von Systemen und eignet sich daher grundsätzlich für langlebige, zukunftsfähige Software.

Frage ist nur: Welche Modelle wären geeignet? Wann werden sie in welcher Form eingesetzt?

Software Engineering Rationale: Wissen über Software erheben und erhalten

Kurt Schneider

Von Anforderungsanalyse bis zu Test und Wartung werden unzählige große und kleine Entscheidungen gefällt und Informationen über Software in diese eingearbeitet. Verliert man sie, so verliert man zunehmend auch die Fähigkeit, die Software zu pflegen. Durch innovative Techniken und Werkzeugen soll es gelingen, dieses Zusatzwissen zu sichern, ohne viel spürbaren Zusatzaufwand zu treiben. Entwurfsbegründungen, Erfahrungen und Feedback aus dem Betrieb helfen dann, die Software beweglich und nützlich zu erhalten.

Software altert relativ zu den veränderten Kontextbedingungen: Mitarbeiter gehen und nehmen ihr Wissen mit. Neue Mitarbeiter kommen und verändern die Software, ohne frühere Entscheidungen und ihre Grundlagen zu kennen. Nur wenn man schnell bemerkt, dass sich da eine Lücke auftut, kann man reagieren und die Veralterung aufhalten. Das gilt nicht nur in der Entwicklung sondern auch später im Betrieb. Zusammen mit dem Programm muss auch Wissen bewahrt werden. Dann kann es sein, dass dieses Wissen in Form von Modellen, Entscheidungen und Erfahrungen wichtiger wird als das Programm selbst. "Die Software" lebt weiter, obwohl das Programm ersetzt wird.

Mit einem Softwareprodukt muss ein Schatz technischer und umgebungsbezogener Informationen wachsen. Aufgebaut wird er mit möglichst wenig Zusatzaufwand, ja vielleicht sogar mit unmittelbarem Zusatznutzen, während der Softwareentwicklung. Es gibt mehrere Beispiele, wie dieses Konzepte umgesetzt werden kann. Zum Beispiel wurde das Werkzeug FOCUS entwickelt: damit kann man durch Demonstrationen Wissen extrahieren, das in einem Programm und seinem Entwickler steckt (ICSE 1996). In Architekturen großer Systeme können Vorkehrungen getroffen werden, um Feedback und Erfahrungen mit wenig Zusatzaufwand zu erhalten.

Ich schlage vor, diese Forschungsrichtung weiter zu verfolgen: Werkzeuge, die sich an die Modelle anlehnen und in die Architektur integrieren, um Wissen zu erheben und zu erhalten. Dahinter sind semantisch weitergehende Aufbereitungen möglich. Zunächst ist es aber am Wichtigsten, an vielversprechenden Stellen in Entwicklung und Betrieb Feedback einzusammeln und Erfahrungen zu erheben - damit sie für langlebigere Konzepte und Software genutzt werden können.

Modellbasierte Architekturevolution verteilter Software-Systeme

Andy Schürr

Modellbasierte und gleichzeitig architekturgetriebene Ansätze der Software-Entwicklung gewinnen unter verschiedenen Namen wie "Model Driven Architecture" (MDA) oder "Model Driven Software Development" (MDSD) in den letzten Jahren in verschiedensten Branchen zunehmend an Bedeutung; gerade Deutschland spielt im internationalen Vergleich dabei eine Vorreiterrolle in der Forschung und im industriellen Einsatz solcher Techniken (z.B. im Automobilbereich). Bisher werden solche modellbasierten Ansätze aber praktisch ausschließlich bei der Neuentwicklung von Anwendungen eingesetzt. Die Modernisierung existierender Alt-Anwendungen geschieht bisher in aller Regel direkt auf dem Quellcode und meist in eher unsystematischer Weise während der Lebenszeit der Anwendung. In automobilen Anwendungen werden beispielsweise neue Anforderungen an Steuergerätefunktionen oft nur durch schrittweise Anpassung und Weiterentwicklung des Codes umgesetzt. Diese Vorgehensweise scheitert, wenn bestehende Funktionalität zerlegt werden soll, um neue Steuergerätearchitekturen oder -topologien zu unterstützen. In der Konsequenz muss der bestehende Code mit hohem Aufwand überarbeitet werden. Diese Aufwände stehen beispielsweise der Automobilindustrie in den nächsten Jahren bevor, wenn mit AUTOSAR flächendeckend eine neue Systemarchitektur eingeführt wird.

Die Anwendung der Konzepte einer modellbasierten Vorgehensweise zur systematischen Architekturevolution kann deshalb erheblich zur Qualitätssteigerung und Kostensenkung bei der Wartung und Weiterentwicklung von Software beitragen. Auf internationaler Ebene wurde zur Untersuchung genau dieser auch aus Sicht der Forschung noch weitgehend ungelösten Fragestellungen eine OMG Task Force "Architecture-Driven Modernization Roadmap" gegründet. Das damit verwandte Thema der "Koevolution von Modellen und Programmcode" ist zudem international ein forschungspolitisch wichtiges Thema, da sich immer mehr die Erkenntnis durchsetzt, dass nicht konsistent gepflegte Modelle in kurzer Zeit nutzlos werden und dass so der hohe Erstellungsaufwand keinem angemessenen Nutzen gegenübersteht. Diese Koevolution umfasst dabei typischerweise Veränderungen des Programmcodes auf Quelltextebene, Veränderungen der Architekturmodelle (bzw. verschiedene Sichten auf die Architektur) und möglicherweise auch die Veränderung des Architekturbeschreibungskonzeptes. Oft unterscheidet man zwischen der Codearchitektur (Technische Software-Architektur - TA) und der davon unterschiedlichen logischen Architektursicht (funktionale Software-Architektur - FA). Derartige unterschiedliche Architekturmodelle müssen im Entwicklungsprozess in sich konsistent und zueinander kompatibel gehalten werden. Außerdem müssen die Architekturmodelle immer kon-

sistente und gültige Sichten auf das tatsächliche Programm darstellen. Die besondere Herausforderung besteht darin, Restrukturierungen von großen Programmsystemen in einem kontinuierlichen und schrittweisen Prozess zu bewältigen, während dessen das Anwendungssystem weiter existiert und im Einsatz bleibt. Im Rahmen von Forschungsaktivitäten am Fachgebiet Echtzeitsysteme der TU Darmstadt werden solche Restrukturierungsprozesse auf der Ebene der Architekturbeschreibungen untersucht und methodisch unterstützt.

Robuste und wiederverwendbare Softwaresysteme durch Komponentenprotokollprüfung

Wolf Zimmermann, Andreas Both

Ein robustes und wiederverwendbares Softwaresystem sollte eine komponentenbasierte oder serviceorientierte Softwarearchitektur haben, da idealerweise durch einfaches Austauschen oder Ergänzen von Komponenten ein Softwaresystem weiterentwickelt werden kann. Dieser Idealzustand ist derzeit nicht erreicht, denn das Zusammensetzen oder Verändern von Komponenten führt häufig zu unerwarteten Effekten. So verhalten sich Komponenten anders als erwartet, wenn sie in einem anderen als dem erwarteten Kontext eingesetzt werden. Es entstehen auf Grund der in vielen Komponententechnologien vorhanden Nebenläufigkeit Verklemmungen auf Grund der Komponentenkomposition und Komponenten stürzen bei ihrer Ausführung ab weil sie nicht vorhersehbar genutzt werden. Ursachen sind u.A., dass die Komposition rein syntaktisch an Hand von Schnittstellen erfolgt und dass Komponenten zustandsbehaftet sind. Letzteres bedingt, dass eine unerwartete Aufrufreihenfolge der Dienste einer Komponente fehlerhaftes Verhalten verursachen kann.

Wir schlagen - wie eine Reihe anderer Arbeiten - vor, dass zusätzlich zu den Schnittstellen Komponenten um Protokolle erweitert werden, die die Menge der zulässigen Aufrufreihenfolgen spezifiziert. Die Benutzung einer Komponente C in einem Komponentensystem ist die Menge tatsächlichen Aufrufreihenfolgen. Eine Komponentenprotokollprüfung für Komponente C prüft konservativ, ob deren Benutzung Teilmenge des Protokolls ist. Konservative Prüfung bedeutet, dass zwar Fehlalarame möglich sind, aber Positivmeldungen auf jeden Fall korrekt sind.

In unserer Arbeit sind die Protokolle reguläre Sprachen und können daher durch reguläre Ausdrücke formuliert werden. Ziel ist eine vollautomatische Protokollprüfung, die nur die Erstellung der Protokolle erfordert. Wir haben gezeigt, dass die von vielen anderen Arbeiten zur Abstraktion des Komponentenverhaltens benutzten endlichen Transduktoren (Übersetzung der Protokolle in reguläre Ausgabesprachen) zu falschen Positivmeldungen führen kann. Ursache sind vor allem rekursive Rückaufrufe an Komponenten. Um sowohl unbeschränkte Nebenläufigkeit als auch unbeschränkte Rekursion im Verhalten von Komponenten und Komponentensystemen zu modellie-

ren, benutzen wir Process Rewrite Systems, die als Spezialfälle Kellerautomaten und Petri-Netze umfassen. Mit dieser Technologie ist eine Protokollprüfung automatisch durchführbar.

Unser Verfahren wird in einer Fallstudie eines mittelständischen Unternehmens auf Praxistauglichkeit angepasst. In diesem Kontext haben wir Protokollprüfungen für Python entwickelt. Derzeit sind Komponentenprotokollprüfungen für C++ und BPEL in Arbeit. Es hat sich zeigt, dass die manuelle Erstellung von Protokollen in der industriellen Praxis machbar ist und sogar schon ohne automatisierten Protokollprüfungen zu Qualitätsverbesserungen führt.

Wir sind überzeugt, dass Protokolle mit automatischen Prüfverfahren einen Beitrag zur Stabilisierung von komponentenbasierten und serviceorientierten Softwarearchitekturen liefern. Im Zusammenhang mit dem Workshop-Ziel stellen sich eine Reihe von wiss. Fragestellungen. So sollte untersucht werden, wie stabil die Protokolle im Laufe der Evolution eines Softwaresystems sind. Eine andere interessante Fragestellung ist, wie die derzeit rein statische Protokollprüfung inkrementell erfolgen kann und evtl. alte erfolgreiche Protokollprüfungen wiederverwenden kann.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2006-04

2006-04	H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel	Handbuch zu MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen
2007-01	M. Conrad, H. Giese, B. Rumpe, B. Schätz (Hrsg.)	Tagungsband Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme III
2007-02	J. Rang	Design of DIRK schemes for solving the Navier-Stokes-equations
2007-03	B. Bügling, M. Krosche	Coupling the CTL and MATLAB
2007-04	C. Knieke, M. Huhn	Executable Requirements Specification: An Extension for UML 2 Activity Diagrams
2008-01	T. Klein, B. Rumpe (Hrsg.)	Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, Tagungsband
2008-02	H. Giese, M. Huhn, U. Nickel, B. Schätz (Hrsg.)	Tagungsband des Dagstuhl-Workshopss MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV
2008-03	R. van Glabbeek, U. Goltz, J.-W. Schicke	Symmetric and Asymmetric Asynchronous Interaction
2008-04	R. van Glabbeek, U. Goltz, J.-W. Schicke	On Synchronous and Asynchronous Interaction in Distributed Systems
2008-05	M. V. Cengarle, H. Grönniger B. Rumpe	System Model Semantics of Class Diagrams
2008-06	M. Broy, M. V. Cengarle, H. Grönniger B. Rumpe	Modular Description of a Comprehensive Semantics Model for the UML (Version 2.0)
2008-07	C. Basarke, C. Berger, K. Berger, K. Cornelsen, M. Doering J. Effertz, T. Form, T. Gülke, F. Graefe, P. Hecker, K. Homeier F. Klose, C. Lipski, M. Magnor, J. Morgenroth, T. Nothdurft, S. Ohl, F. Rauskolb, B. Rumpe, W. Schumacher, J. Wille, L. Wolf	2007 DARPA Urban Challenge Team CarOLO - Technical Paper
2008-08	B. Rosic	A Review of the Computational Stochastic Elastoplasticity
2008-09	B. N. Khoromskij, A. Litvinenko, H. G. Matthies	Application of Hierarchical Matrices for Computing the Karhunen-Loeve Expansion
2008-10	M. V. Cengarle, H. Grönniger B. Rumpe	System Model Semantics of Statecharts
2009-01	H. Giese, M. Huhn, U. Nickel, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme V
2009-02	D. Jürgens	Survey on Software Engineering for Scientific Applications: Reuseable Software, Grid Computing and Application
2009-03	O. Pajonk	Overview of System Identification with Focus on Inverse Modeling
2009-04	B. Sun, M. Lochau, P. Huhn, U. Goltz	Parameter Optimization of an Engine Control Unit using Genetic Algorithms
2009-05	A. Rausch, U. Goltz, G. Engels, M. Goedicke, R. Reussner	LaZuSo 2009: 1. Workshop für langlebige und zukunftsfähige Softwaresysteme 2009