MBEES 2007
MBEES 2007
MBEES 2007
MBEES 2007
MBEES 2007

Tagungsband des Dagstuhl-Workshops

Modellbasierte Entwicklung eingebetteter Systeme III

Mirko Conrad
Holger Giese
Bernhard Rumpe
Bernhard Schätz

[CGR+07] M. Conrad, H. Giese, B. Rumpe , B. Schätz (Hrsg.).
Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III.
Informatik- Bericht 2007-01, Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik,
Technische Universität Braunschweig, 2007.
www.se-rwth.de/publications

Tagungsband

Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III

Model-Based Development of Embedded Systems

15. – 18.01.2007

Informatik-Bericht
2007-01
TU Braunschweig

Institut für
Software Systems Engineering
Technische Universität Braunschweig
Mühlenpfordtstraße 23
D-38106 Braunschweig

Organisationskomitee

Mirko Conrad, The MathWorks

Holger Giese, Univ. Paderborn

Bernhard Rumpe, TU Braunschweig

Bernhard Schätz, TU München

Programmkomitee

Michael von der Beeck, BMW Group

Jörg Desel, KU Eichstätt

Heiko Dörr, Daimler Chrysler AG

Ulrich Freund, ETAS GmbH

Michaela Huhn, TU Braunschweig

Hardi Hungar, OFFIS e.V.

Torsten Klein, Carmeq GmbH

Tiziana Margaria, Univ. Potsdam

Oliver Niggemann, dSPACE GmbH

Alexander Pretschner, ETH Zürich

Holger Schlingloff, Fraunhofer FIRST

Andreas Schürr, Univ. Darmstadt

Albert Zündorf, Univ. Kassel

Inhaltsverzeichnis

Towards Tool Support for Service-Oriented Development of Embedded Automotive Systems
Vina Ermagan, To-Ju Huang, Ingolf H. Krüger, Michael Meisinger, Massimiliano Menarini, Praveen Moorthy 1

From Component Models to Function Nets?
Jan Philipps 25

Obstacles to the Adoption of Model-based Design within the Automotive Supply Industry
Eva Kalix, Oliver Schütte 29

Herausforderungen bei der Neugestaltung von Seriensoftwareentwicklungs-umgebungen
Andy Yap, Peter Großhans 35

Using Simulink® and Real-Time Workshop® Embedded Coder for Safety-Critical Automotive Applications
Mirko Conrad 41

Notation und Verfahren zur automatischen Überprüfung von temporalen Signal-abhängigkeiten und -merkmalen für modellbasiert entwickelte Software
Carsten Gips, Hans-Werner Wiesbrock 51

Automatisierte, werkzeugübergreifende Richtlinienprüfung zur Unterstützung des Automotive-Entwicklungsprozesses
Tibor Farkas, Harald Röbig 61

Typisierung und Verifikation zeitlicher Anforderungen automotiver Software Systeme
Matthias Gehrke, Martin Hirsch, Wilhelm Schäfer, Oliver Niggemann, Dirk Stichling, Ulrich Nickel 73

Das MATE Projekt – visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen
Ingo Stürmer, Heiko Dörr, Holger Giese, Udo Kelter, Andy Schürr, Albert Zündorf 83

Requirements Engineering in der Analysephase mit der Rational Suite
Michael Erskine 95

Simulation-Driven Creation, Validation and Evolution of Behavioral Requirements Models
Martin Glinz, Christian Seybold, Silvio Meier 103

Generierung von UML-Modellen aus formalisierten Anwendungsfallbeschreibungen
Mario Friske, Bernd-Holger Schlingloff 113

Dagstuhl-Workshop MBEES:
Modellbasierte Entwicklung eingebetteter Systeme III
(Model-Based Development of Embedded Systems)

Innovationen brauchen oft genau eine Generation, bis sie vollends in der Industrie eingesetzt werden. Die Konsolidierung der Informatik zeigt, dass auch hier die Innovationsgeschwindigkeit abnimmt und es mittlerweile deutlich länger dauert neue Konzepte und Technologien zur industriellen Reife zu führen und breit in der Praxis zu verankern. Die grundsätzlichen Vorteile einer intensiven Nutzung von Modellen ist mit den Schlagworten „Model-Driven Architecture“ (MDA) und „Model Driven Engineering“ (MDE) mittlerweile deutlich klar geworden. Allerdings ist die Hürde einer erfolgreichen Verankerung in der industriellen Softwareentwicklung immer noch hoch. Die Einstiegskosten zum erfolgreichen Einsatz scheinen manchmal zu hoch, die Technologie noch optimierbar, die Kenntnisse zur Nutzung eines mächtigen, aber gefährlichen Konzepts zu gering.

Bereits in MBEES II (im Januar 2006) wurde festgestellt, dass modellbasierte Entwicklung in der Lage sein muss, Modelle einzusetzen, die sich an der Problem- anstatt der Lösungsdomäne orientieren. Dies bedingt einerseits die Bereitstellung anwendungsorientierter Modelle (z.B. MATLAB/Simulink-artige für regelungstechnische Problemstellungen, Statechart-artige für reaktive Anteile) und ihrer zugehörigen konzeptuellen (z.B. Komponenten, Signal, Nachrichten, Zustände) und semantischen Aspekte (z.B. synchroner Datenfluss, ereignisgesteuerte Kommunikation). Andererseits bedeutet dies auch die Abstimmung auf die jeweilige Entwicklungsphase, mit Modellen von der Anwendungsanalyse (z.B. Beispielszenarien, Schnittstellenmodelle) bis hin zur Implementierung (z.B. Bus- oder Task-Schedules, Implementierungstypen). Für eine durchgängige modellbasierte Entwicklung ist daher im Allgemeinen die Verwendung eines Modells nicht ausreichend, sondern der Einsatz einer Reihe von abgestimmten Modellen für Sichten und Abstraktionen des zu entwickelnden Systems (z.B. funktionale Architektur, logische Architektur, technische Architektur, Hardware-Architektur) nötig.

Durch den Einsatz problem- statt lösungszentrierter Modelle kann in jedem Entwicklungsabschnitt von unnötigen Festlegungen abstrahiert werden. Dafür geeignete Modell-Arten sind weiterhin in Entwicklung und werden in Zukunft immer öfter eingesetzt. Dennoch ist noch vieles zu tun, speziell im Bau effizienter Werkzeuge, Optimierung der im Einsatz befindlichen Sprachen und der Schulung der Softwareentwickler in diesem neuen Entwicklungsparadigma. Diese neuen Modell-Arten und ihre Werkzeuge werden die Anwen-

dung analytischer und generativer Verfahren ermöglichen und damit bereits in naher Zukunft eine effiziente Entwicklung hochqualitativer Software erlauben.

Weiterhin sind im Kontext der modellbasierten Entwicklung viele, auch grundlegende Fragen offen, insbesondere im Zusammenhang mit der Durchgängigkeit. Die in diesen Tagungsband zusammengefassten Papiere stellen zum Teil gesicherte Ergebnisse, Work-In-Progress, industrielle Erfahrungen und innovative Ideen aus diesem Bereich zusammen und erreichen damit eine interessante Mischung theoretischer Grundlagen und praxisbezogener Anwendung.

Genau wie bei den ersten beiden, im Januar 2005 und 2006 erfolgreich durchgeführten Workshops sind damit wesentliche Ziele dieses Workshops erreicht:

- Austausch über Probleme und existierende Ansätze zwischen den unterschiedlichen Disziplinen (insbesondere Elektro- und Informationstechnik, Maschinenwesen/Mechatronik und Informatik)
- Austausch über relevante Probleme in der Anwendung/Industrie und existierende Ansätze in der Forschung
- Verbindung zu nationalen und internationalen Aktivitäten (z.B. Initiative des IEEE zum Thema Model-Based Systems Engineering, GI-AK Modellbasierte Entwicklung eingebetteter Systeme, GI-FG Echtzeitprogrammierung, MDA Initiative der OMG)

Die Themengebiete, für die dieser Workshop gedacht ist sind fachlich sehr gut abgedeckt, auch wenn sie sich auch dieses Jahr (mit Ausnahmen) sehr stark auf den automotiven Bereich konzentrieren. Sie fokussieren auf Teilaspekte modellbasierter Entwicklung eingebetteter Softwaresysteme. Darin enthalten sind unter anderem:

- Domänenspezifische Ansätze zur Modellierung von Systemen
- Durchgängiger Einsatz von Modellen
- Modellierung spezifischer Eigenschaften eingebetteter Systeme (z.B. Echtzeit- und Sicherheitseigenschaften)
- Konstruktiver Einsatz von Modellen (Generierung)
- Modellbasierte Validierung und Verifikation

Das Organisationskomitee ist der Meinung, dass mit den Teilnehmern aus Industrie, Werkzeugherstellern und der Wissenschaft die bereits 2005 und

2006 erfolgte Community-Bildung erfolgreich weitergeführt wurde, und damit demonstriert, dass eine solide Basis zur Weiterentwicklung des sich langsam entwickelnden Felds modellbasierter Entwicklung eingebetteter Systeme existiert.

Die Durchführung eines erfolgreichen Workshops ist ohne vielfache Unterstützung nicht möglich. Wir danken daher den Mitarbeitern von Schloss Dagstuhl und natürlich unseren Sponsoren.

Schloss Dagstuhl im Januar 2007,

Das Organisationskomitee

Mirko Conrad, The MathWorks

Holger Giese, Univ. Paderborn

Bernhard Rumpe, TU Braunschweig

Bernhard Schätz, TU München

mit Unterstützung von

Holger Krahn, TU Braunschweig

The MathWorks ist der weltweit führende Anbieter von Technical Computing und Model-Based Design Software für Ingenieure und Wissenschaftler in der Industrie, Forschung und Lehre. Mit einer breit aufgestellten Produktfamilie, die auf den Kernprodukten MATLAB und Simulink basiert, bietet The MathWorks Entwicklungswerkzeuge und Dienstleistungen zur Lösung anspruchsvoller technischer Problemstellungen und zur schnelleren Umsetzung von Innovationen für die Bereiche Automobil, Luft- und Raumfahrt, Telekommunikation, Halbleiter-/Elektrotechnik, Industrielle Automation und Maschinenbau, Medizin, Finanzwesen, Biotechnologie und für weitere Branchen. The MathWorks, mit Hauptsitz in Natick, Massachusetts (USA), wurde 1984 gegründet und beschäftigt über 1400 Mitarbeiter weltweit.

Mit der Vision durch Optimierung der Entwicklungsprozesse fehlerfreie eingebettete Software zu entwickeln, wurde das Unternehmen IT Power Consultants gegründet. IT Power Consultants bietet überzeugende Kernkompetenzen hinsichtlich der Entwicklungsprozesse von Steuergeräte- und eingebetteter Software sowie bezüglich der Methoden und Tools für deren Qualitätssicherung. Das Expertenwissen und die einschlägigen Erfahrungen auf diesen Gebieten vereint IT Power Consultants zu innovativen Dienstleistungen und Produkten für die Software-Entwicklung in der Automobilindustrie. IT Power Consultants wurde im Juni 2000 von Dr.-Ing. Sadegh Sadeghipour und Dipl.-Inf. Mansour Kalantary in Berlin gegründet und beschäftigt derzeit 10 Mitarbeiter.

Innerhalb der Gesellschaft für Informatik e.V. (GI) befasst sich eine große Anzahl von Fachgruppen explizit mit der Modellierung von Software- bzw. Informationssystemen. Der erst neu gegründete Querschnittsfachausschuss Modellierung der GI bietet den Mitgliedern dieser Fachgruppen der GI - wie auch nicht organisierten Wissenschaftlern und Praktikern - ein Forum, um gemeinsam aktuelle und zukünftige Themen der Modellierungsforschung zu erörtern und den gegenseitigen Erfahrungsaustausch zu stimulieren.

Das Institut für Software Systems Engineering (SSE) der TU Braunschweig entwickelt einen innovativen Ansatz des Model Engineering, bei dem ein Profil der UML entwickelt wird, das speziell zur Generierung modellbasierter Tests und zur evolutionären Weiterentwicklung auf Modellbasis geeignet ist (B. Rumpe: Agile Modellierung mit UML. Springer Verlag 2004). SSE ist auch Mitherausgeber des Journals on Software and Systems Modeling.

Der Lehrstuhl für Software Systems Engineering der TU München entwickelt in enger Kooperation mit industriellen Partnern modellbasierte Ansätze zur Entwicklung eingebetteter Software. Schwerpunkte sind dabei die Integration ereignisgetriebener und zeitgetriebener Systemanteile, die Berücksichtigung sicherheitskritischer Aspekte, modellbasierte Testfallgenerierung und modellbasierte Anforderungsanalyse, sowie den werkzeuggestützten Entwurf.

Das Software Quality Lab (s-lab) ist ein Institut für Kompetenz- und Technologietransfer, in dem Partner aus der industriellen Softwareentwicklung mit Forschungsgruppen der Universität Paderborn auf dem Gebiet der Softwaretechnik eng zusammenarbeiten. Zielsetzung des s-lab ist die Entwicklung und Evaluierung von konstruktiven und analytischen Methoden sowie Werkzeugen der Softwaretechnik, um qualitativ hochwertige Softwareprodukte zu erhalten. Eine hohe Relevanz für die industrielle Softwareentwicklung sowie die Notwendigkeit des Einsatzes wissenschaftlicher Methoden kennzeichnen die im s-lab bearbeiteten Fragestellungen.

Schloss Dagstuhl wurde 1760 von dem damals regierenden Fürsten Graf Anton von Öttingen-Soetern-Hohenbaldern erbaut. 1989 erwarb das Saarland das Schloss zur Errichtung des Internationalen Begegnungs- und Forschungszentrums für Informatik. Das erste Seminar fand im August 1990 statt. Jährlich kommen ca. 2600 Wissenschaftler aus aller Welt zu 40-45 Seminaren und viele sonstigen Veranstaltungen.

Towards Tool Support for Service-Oriented Development of Embedded Automotive Systems

Vina Ermagan[1], To-Ju Huang[1], Ingolf H. Krüger[1],
Michael Meisinger[2], Massimiliano Menarini[1], Praveen Moorthy[1]

[1] Department of Computer Science
University of California, San Diego
La Jolla, CA 92093-0404, USA
{vermagan,t3huang,ikrueger,mamenari,pmoorthy}@cs.ucsd.edu

[2] Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
meisinge@in.tum.de

Abstract: The development of embedded systems is a challenging task because of the distributed, reactive and real-time nature of such systems. Distribution of embedded components across buses and networks causes high interaction complexity. We propose a model-based development approach to handle this complexity. We model the individual functionalities of the system – the services – independently from each other in an interaction modeling and architecture definition language. Methodological steps allow us to refine and modify the models. A development process determines the order in which to perform the steps. Our service-oriented development methodology spans the entire development process from requirements analysis to implementation, verification and validation. We have developed integrated tool support that governs this process; it provides an effective means to apply and evaluate our approach. In this paper we introduce our service-oriented methodology and describe our tool as part of an integrated tool suite supporting this process by means of an automotive example: the Central Locking System (CLS).

1 Introduction

Designing complex distributed systems is a difficult task. Systems like these can be found in application domains such as avionics and automotive control systems – in the form of embedded systems – and in telecommunications and business information systems. Also sensor networks and mobile applications are growing areas for complex distributed systems. The common property among all of these systems is that distribution and complex interactions between distributed nodes are key enablers of their success. Distribution makes the system architecture much more modular and decentralized, which contributes to fault tolerance, component reuse, system robustness, maintainability and further positive system properties.

However, heavily distributed systems are among the most complex man-made artifacts known; the high degree of distribution makes development of these systems very challenging. The number of states and conditions – for functional behavior as well as for error conditions – increase exponentially with the number of distributed nodes. Furthermore, logistics and maintenance problems originate from the distribution of entities. In fact, nodes can be developed, maintained, extended and replaced independently, potentially harming the overall system integrity and consistency.

1.1 Problem Definition

Important questions that need to be addressed when designing distributed systems and their architectures include, for instance, what is the functional behavior of the system i.e. what are the individual services that the system and its parts offer to the environment and how are services connected to provide the full system functionality. Important further questions are how to design the objects/components of the system and their interfaces so that they can provide the identified services with the required quality properties; how to connect and distribute them i.e. how to design the communication topology, and how to replicate components for most efficient operation on a given middleware. Answering these questions requires a systematic, iterative approach in system design. Often, this involves changes to the designed system behavior and consideration and exploration of different alternative architecture candidates that can provide the designed services.

Model-based development is a promising approach to mitigating the difficulties and complexities of developing a system from requirements analysis to system execution [Bro05]. Models are abstractions of reality that are targeted to express specific views on the system serving a specific purpose. Different models exist at different stages of the development process. Having a collection of complementing models, it is possible to understand, design and modify concepts that are otherwise too complex to handle. If the models used for system development are integrated and consistent, and if there is a systematic process from models of high abstraction through refinement to the realized system, we speak of model-based development. The degree of precision, the level of formality and the used modeling notations and concepts vary significantly from one model-based approach to another.

One model-based approach to designing distributed embedded systems is to separate an overall system model into logical models (also called *domain models* [Eva03]) and implementation models; approaches advocating this separation are architecture-centric software development [OMG05] and model-driven architecture [OMG03]. The logical models describe functionality, distribution of components and quality properties independently of implementation details and deployment architecture design decisions. The implementation models are consistent refinements of the logical models and contain these details and decisions. The advantage of this separation is that one logical model can be refined into many implementation models. This provides an independence of system functionality from the actual deployment architectures, for instance to defer design decisions or to switch to a different architecture.

A clear separation into logical and implementation models is often difficult to achieve – especially in situations where requirements suggest a tight coupling between the two types of models. This is often the case when requirements include specific performance and other Quality-of-Service properties. A core source of complexity is that the scenarios supported by the system typically involve a multitude of collaborating entities partaking in complex interactions. These interactions are part of both the logical and implementation models. A well-defined mapping needs to exist between the interactions in both models.

Our goal is to provide a solution where a logical system behavior model can be reused unchanged across all implementation models (or target-architectures). The major step toward achieving this goal is to decouple the "features" or "services" a system provides from the architecture on which it is deployed.

A development approach must be supported by efficient tools to provide a practical application environment. The methodology establishes the formal and logical basis on top of which tools need to leverage the practical problems and day-to-day routines. Tools automate methodological steps and enable graphical, iterative system modeling, development and verification. Different tools need to be integrated in order to provide seamless modeling support, a high degree of automation and short turn-around times.

1.2 Service-Oriented Specifications

In this paper, we propose an approach to service-oriented development of distributed embedded systems that establishes a clean separation between the services provided by the system under consideration, and the architecture – comprised of components and their relationships – implementing the services. Our approach is well suited for tool support, which we will explain subsequently.

We use the notion of *service* to decouple abstract behavior from implementation architectures supporting it. The term "service" is used in multiple different meanings and on multiple different levels of abstraction throughout the Software Engineering community [TRH+04]. *Web Services* [STK02] currently receive a lot of attention from both academia and industry, but services are also emerging in embedded systems. Fig. 1 shows a typical "layout" of applications composed of a set of services. Often such systems consist of at least two distinct layers: one *domain layer*, which houses all domain objects and their associated logic; and one *service layer*, which acts as a facade to the underlying domain objects – in effect offering an interface that shields the domain objects from client software. For an embedded system, the service layer, for instance, consists of the functions this system exposes to the environment by exposing an access interface on a broadcast bus network, such as CAN and FlexRay. The domain layer consists of "plain old objects" representing the data and logic of the underlying implementation. Typically, services in this sense coordinate workflows among the domain objects; they may also call, and thus depend on, other services. Some of the services, say *Service 1* and *Service 2* in our example, may reside on the same electronic control unit (ECU), whereas others, such as *Service n* may be accessible remotely via the bus network.

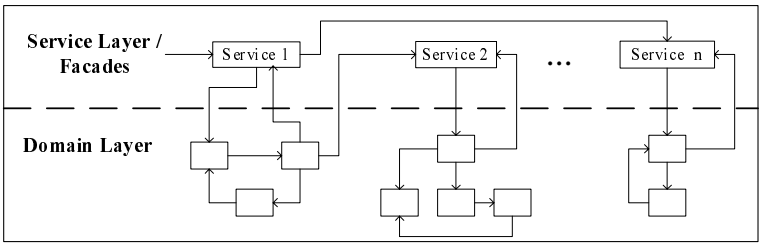


Figure 1: Service-Oriented Architectures

The layout shown in Fig. 1 is prototypical for many domains where complex, often distributed applications are expected to offer externally accessible interfaces. Indeed, service-oriented approaches to system development, leading to similar application structures, are prominent for business information systems using web services [STK02] and in the telecommunications domain [Zav01] and are emerging in the automotive domain [BKM06a, BKM06b]. Abstracting from the domain-specific details we observe that services often encapsulate the coordination of sets of domain objects to implement "use cases".

We view services as specializations of use cases to specify interaction scenarios; services "orchestrate" the interaction among certain entities of the system under consideration to achieve a certain goal [Eva03]. In contrast to use cases, which describe functionality typically in prose and on a coarse level of detail, we define a service via the interaction pattern among a set of collaborators required to deliver the functionality. Services are partial interaction specifications. For a formal definition of the service notion, see [KMLS05].

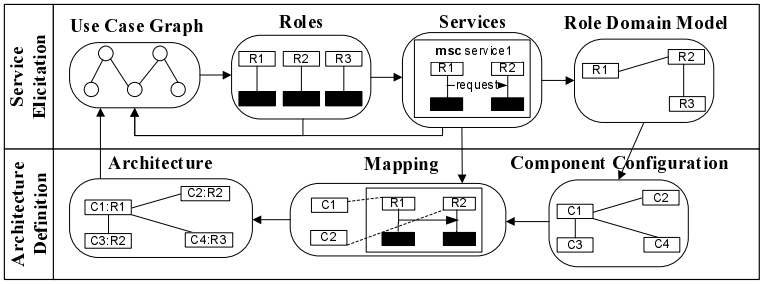


Figure 2: Service-Oriented Development Process

We employ a two-phase, iterative development process as shown in Fig. 2. Phase (1), *Service Elicitation*, consists of defining the set of services of interest – we call this set the service repository, our logical model. Phase (2), *Architecture Definition*, consists of mapping the services to component configurations to define deployments of the architecture – the implementation model.

In phase (1) we identify the relevant use cases and their relationships in the form of a use case graph. This gives us a relatively large-scale, scenario-based view on the system.

From the use cases, we derive sets of *roles* and *services* as interaction patterns among roles. Using roles decouples from interaction details, because roles abstract from components or objects. Roles describe the contribution of an entity to a particular service independently of what concrete implementation component will deliver this contribution. An object or component of the implementation typically will play multiple roles at the same time. The relationships between the roles, including aggregations and multiplicities, develop into the *role domain model*.

In phase (2) the role domain model is refined into a *component configuration*, onto which the set of services is *mapped* to yield an *architectural configuration*. These architectural configurations can be readily implemented and evaluated as target architectures for the system under consideration.

The process is iterative both within the two phases, and across: Role and service elicitation feeds back into the definition of the use case graph; architectures can be refined and refactored to yield new architectural configurations, which may lead to further refinement of the use cases.

1.3 Contributions and Outline

As main contribution, this work presents a systematic approach to the development of distributed, reactive embedded systems and an integrated tool suite we have developed that supports this approach throughout the entire development cycle. We present the application of our modeling approach and the accompanying chain of tools by means of a running example through all stages of development.

In particular, we explain the purpose and depencencies of the individual tools: We use our SODA tool to track requirements of service-oriented systems and generate process required documentation. Our M2Code tool models interaction patterns that define services. The Component Synthesizer of M2Code generates state machines for the modeled services. These state machines can then be transformed into executable code (using RT CORBA CodeGen and M2Aspects), and verified for correctness against the specification (using S2Promela, ServiceDebug and MSCCheck).

In Sect. 2, we introduce the automotive Central Locking System (CLS) as our running example and show how it is modeled in terms of services. In Sect. 3, we show how we make use of tools to support our approach. We present our integrated tool landscape and explain the relevance of specific tools in the process. In Sect. 4, we report on experiences for CLS applying our approach and the tools; we also provide a brief discussion. In Sect. 5 we show related work. Sect. 6 contains conclusions and an outlook.

2 Service-oriented model of CLS

To demonstrate our approach, we use the Central Locking System (CLS), a well-studied and documented example of one automotive vehicle functionality. The CLS integrates a multitude of separate subsystems in the vehicle, ranging from safety critical ones (motor control and crash sensors) to comfort functions (automatic seat positioning and tuner presets in luxury vehicles). For reasons of brevity, we present a simplified and abstract adaptation of the CLS. We direct the reader to [NP03, KNP04] for a more comprehensive description. Here, we focus on some specific use cases during the locking and unlocking of the vehicle: *operation of locks*, *signaling*, *transfer driver ID*, and *impact sensing*.

The Central Locking System in the described form acts as a representative for similar problems in automotive control electronics and distributed, reactive systems in other application domains. For instance, business information systems with distributed components communicating via web services, and database systems implementing distributed transactions by two-phase commit protocols share many of the same properties and challenges; thus, the example and approach presented in the following, provide telling insight for these domains as well.

The first step in the service-oriented process is analyzing the requirements, which leads to a number of use cases and actors. The main parts of the CLS system are a remote key fob and a controller within the car, which receives the lock and unlock command signals from the key fob. The controller also interacts with the lighting system in order to operate the lights, the security module, in order to validate a driver's identity by checking the key fob's secure identity token, and the door locking subsystem in order to lock and unlock the doors. In addition, upon impact, an impact sensor will send a signal to the controller. For simplification, we will abstract away the complications of the door locking subsystem by introducing a Lock Manager, which will act as an interface for locking or unlocking the doors.

In the following, we explain in detail how to specify the CLS in our service-oriented modeling approach. We follow our development process, introduced above, to define, implement and verify architectures for distributed, reactive systems. Along the way we introduce our Architecture Definition Language (Service-ADL) that we use for specifying roles, services and architectures; the details of this notation are documented in [KM04, Mat04].

After capturing the use cases and actors, the first step towards a service-oriented system specification is to identify the participating roles, which emerge out of the found actors within the use cases. We identify key fob (KF), controller (CONTROL), lock manager (LM), security module (SM), lighting system (LS), database (DB), which holds the information for each driver ID, and the impact sensor (IS). These roles are the logical entities in our system that communicate locally or over the network to provide the required system functionalities. Fig. 3 shows the role definitions in our Service-ADL.

The next step in our process is to elicit the services that the system needs to support based on the found use cases. For instance, the service *Vehicle Unlocking* involves the communication between the key fob and the controller, which, in turn, communicates with the lock

```
role CONTROL
  description
    Entity acting as a controller for all CLS functions
  states
    (UNLD: Doors unlocked) (LCKD: Doors locked)

role KF
  description
    Represents the receiver unit for a driver's key fob
  states
    (INITIAL: initial state)

role DB description Database component ...
role IS description Impact sensor ...
role LM description Lock manager ...
role LS description Lighting system ...
role SM description Security module ...
```

Figure 3: Role Definitions in Service-ADL

manager (for physical door unlocking), the lighting system (for flashing the lights) and the security module (to validate and store the driver id). Fig. 4 depicts the communication connections between the roles for all services in the CLS example, as part of the service repository definition in Service-ADL.

```
service repository CLS
 description
  Central Locking System
 roles
  (KF)(CONTROL)(SM)(LM)(DB)(IS)(LS)
 services
  (CLS-0)(CLS-1)(UNLK-e)(LCK)(LCK-1)
  (LCK-2)(UNLK)(UNLK-1)(UNLK-2)
 role domain model
```

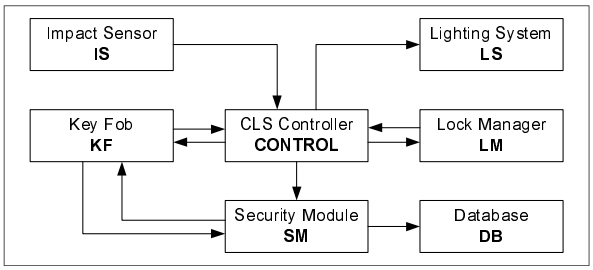


Figure 4: CLS Service Repository Definition

We specify the services of the CLS system using a notation based on Message Sequence Charts (MSC) [IT96, Krü00, OMG05]. An MSC defines the relevant sequences of *messages* (represented by labeled arrows) among the interacting *roles*. Roles are represented as vertical axes in our MSC notation. Fig. 6 and 7 show the specification of several services as interaction patterns. The MSC syntax we use should be fairly self-explanatory, especially to readers familiar with UML2 [OMG05]. In particular, we support labeled boxes in our MSCs indicating alternatives and conditional repetitions (as bounded and un-

bounded loops). Labeled boxes *on* an axis indicate actions, such as local computations; diamond-shaped boxes on an axis indicate state labels. High-level MSCs (HMSCs) indicate sequences of, alternatives between and repetitions of services in two-dimensional graphs – the nodes of the graph are references to MSCs, to be substituted by their respective interaction specifications. HMSCs can be translated into basic MSCs without loss of information [Krü00]. Fig. 5 shows the use of HMSCs in the CLS example. The left HMSC shows the infinite sequence of locking followed by unlocking back to locking, and the right side shows an HMSC defining the *Vehicle Unlocking (UNLK)* service as a composition of two sub-services.

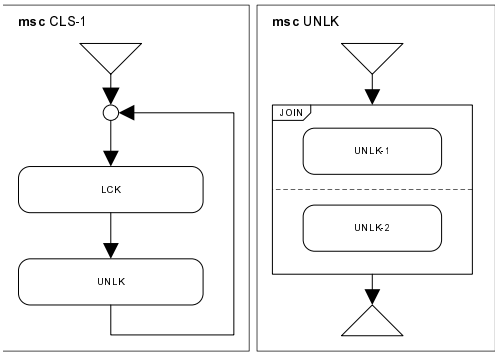


Figure 5: CLS High Level Services CLS-1, LCK

We model the CLS *Vehicle Unlocking* service by joining two modular sub-services (interaction patterns): UNLK-1 and UNLK-2. Fig. 6 shows the UNLK-1 service, which captures the *Operation of Locks and Signaling*. Upon receipt of the *unlck* message from KF, CONTROL issues an *unlck* message to LM. Once LM acknowledges this with an *ok* message, CONTROL requests signaling of the unlocking from LM by means of a *door_unld_sig* message. Once it has issued this message, CONTROL sends an *ok* message back to KF. The *Transfer Driver ID* service – storing the driver's id for further access – is also triggered by the *unlck* message from KF to CONTROL, and is captured in UNLK-2. The corresponding interaction pattern is shown in Fig. 7 as part of a screenshot of our service modeling tool *M2Code*. In a subsequent iteration of the development process we could use the *SODA tool* (see Sect. 3.2) to capture the requirement that a failed security check will *not* unlock the vehicle; consequently the UNLK-1 service could then be modified to include an interaction with SM *before* the locks are operated.

A number of extensions to the standard MSCs warrant explanation [Krü03, KM04]. First, we take each axis to represent a *role* rather than a class, object, or component. The mapping from roles to components is a design step in our approach and will be described below. Furthermore, we use an operator called *join* [Krü00, Krü03], which we use extensively to compose *overlapping* service specifications. We call two services *overlapping* if their interaction scenarios share at least two roles and at least one message between shared roles. The join operator will *synchronize* the services on their shared messages, and otherwise result in an arbitrary *interleaving* of the non-shared messages of its operands.

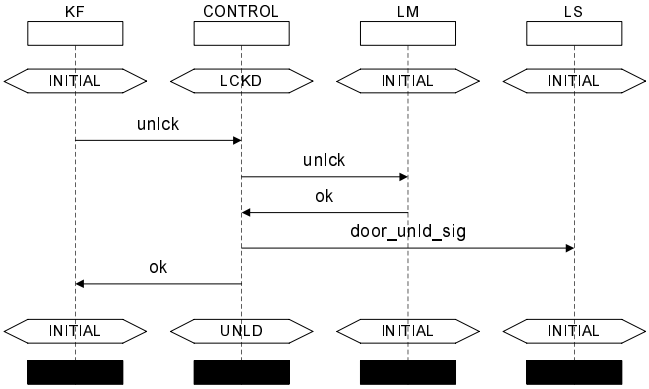


Figure 6: UNLK-1: Operation of Locks & Signaling

Join is a powerful operator for separating an overall service into interacting sub-services. The availability of such an operator also distinguishes our approach from many others in literature. For instance, we use the join operator (as seen on the right side of Fig. 5) to compose the described two unlocking services from Fig. 6 and 7.

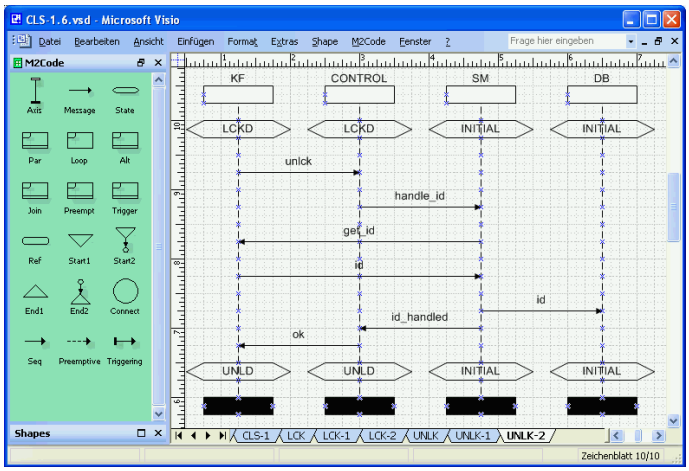


Figure 7: UNLK-2: Transfer Driver ID (screenshot)

To specify preemptive behavior, we introduced the *preempt* operator [Krü00, Krü03]. For CLS, the vehicle must also unlock in case of a crash impact. Upon impact, IS will send an impact signal to CONTROL. At this point, the routine interactions should be preempted, and CONTROL should immediately unlock the doors by sending an unlock message to LM and receiving its acknowledgment. Fig. 8 gives an example for the use of the preempt operator. Both triggering multiple services with the same message and preempting a set

of such complex services with another critical service are powerful capabilities of our approach, as shown in the example.

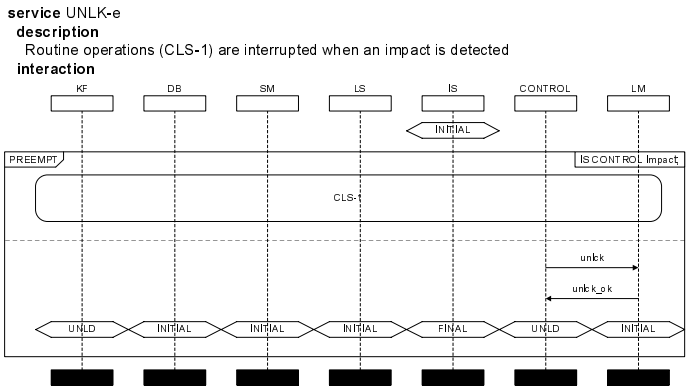


Figure 8: Adding Emergency Unlocking using Preemption

So far, we modeled the logical behavior of the system via services, independently of any underlying deployment architecture. This approach maximizes flexibility and reusability of the logical model. The next step after eliciting the services is to define a suitable component architecture that can provide these services and to define a mapping from the logical model to the deployment model, which includes the mapping of roles to components. We must make sure that the architecture observes the dependencies of the roles and further constraints given by the requirements. Fig. 9 shows how we specify components in our Service-ADL. When a role is mapped to a component, that component plays the mapped role. Intuitively, "playing a role" in an architecture means implementing all interactions in which this role partakes. Multiple roles can be mapped to the same component. The more roles it plays, the more functionality it implements. Interactions between different components usually mean expensive distributed communications, while interactions between roles within one component can be implemented very efficiently as subroutine or method calls. Also, we can map the same role to multiple components, indicating a replication of that role.

```
component type ImpactSensor
  description A sensor to detect an impact
  plays (IS)
  in service (UNLK-e)

component type MainControl
  description The main controller of the CLS
  plays (CONTROL)
  in service (UNLK-1)(UNLK-2)...
...
```

Figure 9: CLS Component Type Definitions

Fig. 10 shows a potential deployment architecture for our CLS case study, defined in our Service-ADL. As shown in the figure, the IS role is played by two different components, meaning that it is replicated, due to its criticality, in order to improve reliability. Also part of the architecture configuration is the mapping of role communication channels to networks, in our case a "Wireless" network and a "CAN Bus". Later in this paper, we will use this deployment architecture to demonstrate how we can verify architectures for distributed reactive systems. For more information about the architecture mapping and architecture exploration, see [KMM06].

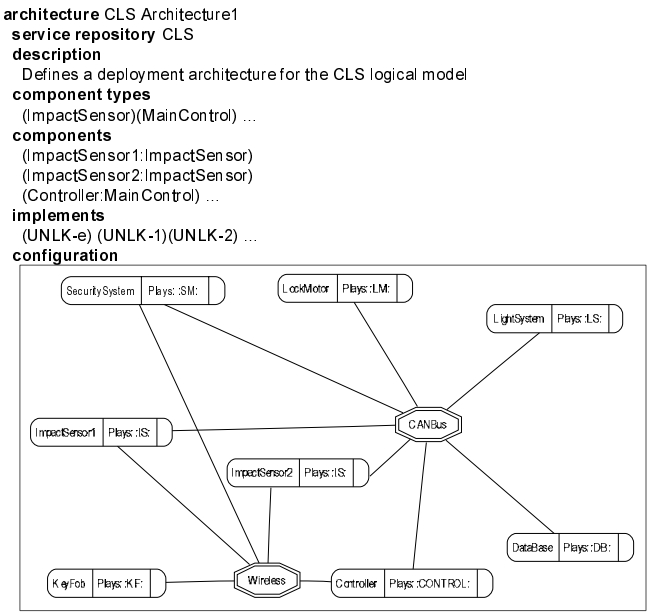


Figure 10: CLS Deployment Architecture

Based on the service repository specification and the deployment architecture definition, we can now simulate the model, verify and model check it, and generate code for prototypical implementations.

3 Integrated Tool Support

In order to demonstrate and experiment with service-based development, we have designed and implemented several novel prototypical tools. Together they form an integrated tool landscape or tool chain. The purpose of this tool chain is to illustrate and support the complete development cycle, from the initial modeling phase to execution on real systems.

3.1 Tool Landscape Architecture

The tool landscape consists of a number of tools supporting interaction-based and service-oriented development. Fig. 11 gives an overview of some of our tools and their dependencies. We divide the tool landscape roughly by the stage in the overall development cycle: *Requirements Elicitation*, *Service Specification and Architecture Design*, *Implementation*, as well as *Deployment and Verification*. In the following sections, we will describe the tools according to their location in the development stage; for a few of the tools we will present more extensive descriptions.

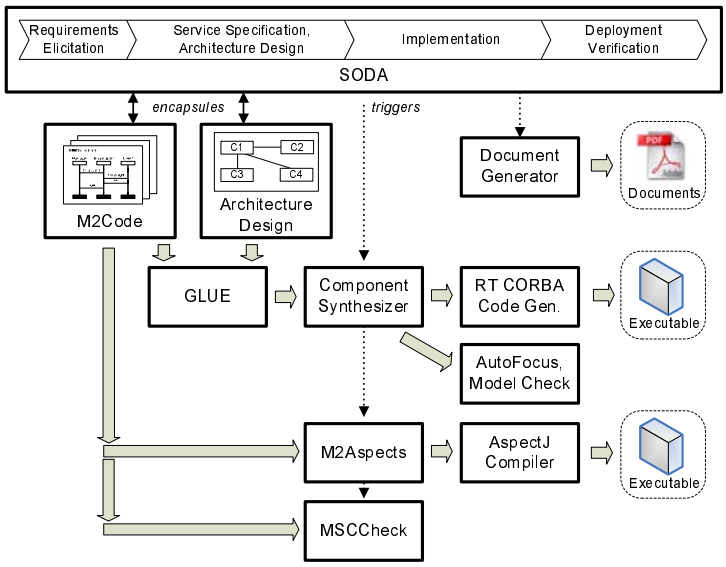


Figure 11: Service-Oriented Tool Landscape

The overall dependencies are as follows: The *SODA tool* tracks requirements of service-oriented systems and generates requirements and architecture documentation. *M2Code* provides facilities for intuitive graphical service modeling with MSCs and HMSCs. M2Code also contains a *Component Synthesizer* to generate state machines for all components partaking in the modeled services. The GLUE tool maps the component state machines to more complicated target architectures, for instance with replicated component instances. The component state machines are the basis for code generation of executable prototypes (using RT CORBA CodeGen and M2Aspects) and verification by model checking (using S2Promela, ServiceDebug and MSCCheck).

3.2 Tools for Requirements Elicitation

We support the process of *Requirements Elicitation* by means of two tools: the *SODA tool* and *M2Code*.

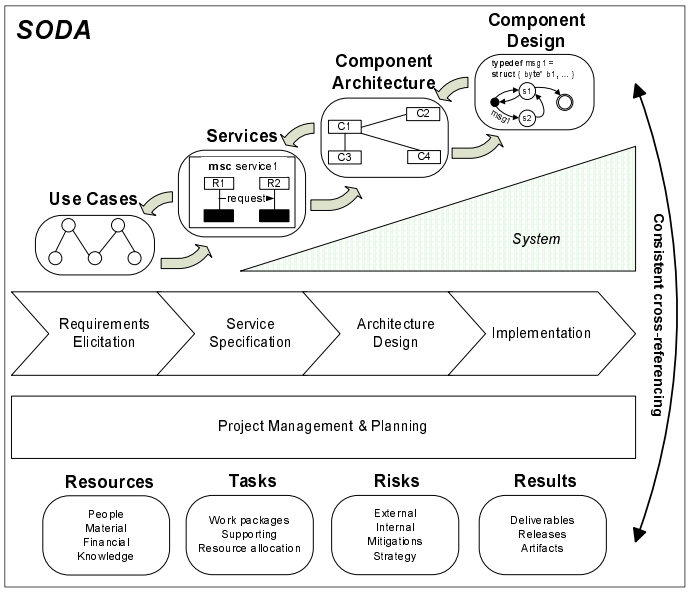


Figure 12: SODA Tool Overview

The *SODA tool* (Service-Oriented Design and Architecture Development) governs the development process and provides project support. It has facilities for managing project resources and tracking project progress. *SODA* embeds and triggers other tools and has the capability of generating project relevant documentation and reports. *SODA tool* is a platform for capturing and cross-referencing information about a system or software under development (see Fig. 12 for a schematic overview). Information is structured according to a changeable domain model and is stored as XML with associated data files. *SODA* contains a generic XML editor that enables editing data documents based on arbitrary XML schemas. We designed an XML schema that contains elements for service-oriented development projects, such as "Use Case", "Textual Requirement", "Service", "Component" and many more. The schema designer can further structure and specify types and attributes for each of these elements Based on the type of element or attribute, specific data editors are invoked. *SODA* provides a powerful linking and cross-referencing facility that applies to all elements, as defined in the XML schema. *SODA* provides powerful mechanisms for editing, linking and post-processing.

In our reference data model for service-oriented development projects, the system developer enters requirements as hierarchically structured use cases, which contain free form text and embedded figures. Use cases are linked to structured textual requirements; both use cases and requirements are additionally linked to the services of the system in devel-

opment. All the before mentioned object types and their cross-references are defined in the underlying XML schema.

SODA has a plug-in architecture that enables consistency checks of the currently edited data model. For instance, *SODA* can detect referencing errors (such as link cycles) and missing references (such as a use case with no linked service). *SODA* also has powerful plug-ins for generating LaTeX, PDF, HTML and other type documents from the model. We use this to generate requirements and software architecture documents out of the data model.

The second tool supporting requirements elicitation is *M2Code*. It is our service modeling tool and will be described in detail in the next section. For requirements gathering we use high level services - in form of HMSCs - without providing detailed service specifications and interaction patterns.

We are currently developing the link between *SODA* and *M2Code*, so that services can be edited directly and stored within *SODA*'s system data model.

3.3 Tools for Service Specification and Architecture Design

A central part of our tool landscape are the modeling and specification tools. *M2Code* enables the system designer to specify functionalities as a complex interplay of logical components (called *Roles*) exchanging messages. Each system functionality is modeled separately from the others. Those functionalities are called "services" in our terminology. Services can be structured hierarchically and referenced from other services. This enables us to design service-oriented applications in a modular way, which results in reusable service definitions. Simpler services are combined to form more complex ones providing advanced functionalities by means of composition operators.

M2Code (see Fig. 7 for a screenshot) is the principal modeling tool of our tool set. It allows modelers to specify system functions (services) using Message Sequence Charts (MSCs). Additional information, such as real time constraints, logical to deployment mappings, failure hypothesis, etc. can be specified using other tools to enrich the model specified by M2Code.

The tool is implemented as a Microsoft Visio plug-in. This architecture allows us to leverage the powerful design and export capabilities of Visio and include interaction specifications in various types of documents. The plug-in provides graphical elements to represent Message Sequence Charts, High Level Message Sequence Charts (HMSCs) and the various operators we use. Fig. 7 for instance, depicts an MSC designed with M2Code. Apart from the graphical capabilities that enable us to design specifications with MSCs and HMSCs, M2Code contains the *Component Synthesizer* to generate state machines and role structure diagrams out of specifications.

We make use of role structures and state machines for the individual components [KGSB99, AKMP05] in other tools for various purposes. The role structure captures the communication links between the roles participating in the specified services.

The generated state machines, one for each role, capture the local behavior of each participant. They define how each node of a distributed system engages with its environment to carry out the specified services. Role structures and state machines are exported to an XML file.

To demonstrate the use of our tools in addressing the needs of the Automotive industry we have used M2Code to model the CLS example described in Sect. 2. We advocate an iterative refinement process that must be supported by powerful language constructs and adequate tools to be effective. Fig. 8 exemplifies how M2Code supports iterative service-oriented development: we use the *PREEMPT* operator to enrich an initial service specification (defined in the referenced service *CLS-1*, see Fig. 5) with emergency unlocking behavior (defined in the lower part of the graph). The parameter on the right part of the *PREEMPT* box specifies that a message "*Impact*" can be sent by the role *IS* to the role *CONTROL* at any moment. If the message is sent, the normal interaction defined by *CLS-1* is interrupted and replaced by the one defined in the lower part of the *PREEMPT* box. In the example if the "Impact" message is sent the car doors unlock.

This simple example shows how, with the right set of tools and languages, service-oriented development can be supported even in application domains not supported by service-oriented middleware. Once the two services are composed, M2Code will then take care of synchronizing all roles participating in the interaction and ensure that the specified logical behavior will be observed in the generated state machines.

M2Code can directly generate state-machines out of service-oriented specifications if the target deployment architectures are simple and roles are mapped one-to-one to executable components. In case of more complex architectures, we have designed the *GLUE tool* [Rus06], which takes the service specifications of M2Code as an input, together with the deployment architecture specification in Service-ADL (see Fig. 9 and 9). *GLUE* will then generate MSCs for each component instance with unique message identifiers, for further processing by a code generator etc.

3.4 Tools for Implementation

We have implemented two tools that take a service-oriented specification and generate executable prototypical implementations out of it: *RT CORBA CodeGen* and *M2Aspects*. *RT CORBA CodeGen* is a template-based code generator. Currently it targets mainly the RT CORBA [Obj02] infrastructure. However, the flexible template-based architecture enables easy ports to other infrastructures and middlewares. We utilize the code generator to create prototypes of our architectures for simulation and validation. This includes runtime verification of Quality of Service (QoS) properties [AKMP05]. In contrast to other simulation tools we have at our disposal, for instance AutoFocus [Aut06], the RT CORBA based runtime system we have implemented provides monitoring and validation mechanisms for *both* logical flow *and* real-time properties.

The code generator itself is written in Java and uses an XML file generated by M2Code as input. To bridge the gap between the abstract XML specification and the executable code,

we have developed a runtime library that uses the facilities provided by the RT CORBA platform. Our runtime system was intentionally kept simple and straightforward by using the Real-Time Event Service (RTES) messaging facility. The RTES provides the fundamental abstraction for asynchronous message passing, enabling each component to operate independently. We have also made use of hooks the RTES provides to incorporate a real-time scheduler. The Time Service provides a distributed, synchronized, global clock to all components in the system. Mechanisms for globally synchronized clocks exist on many embedded platforms and so the runtime system is easily portable. For instance, we could have an implementation that targets a CAN (Controller Area Network) bus for embedded applications and another targeting an enterprise service bus for large federated corporate networks.

The code generator currently provides two templates, supporting two different execution models: *Synchronous* and *Asynchronous*. Both generate C++ code for the RT CORBA platform. The Synchronous execution model operates in a time-synchronous mode; messages are exchanged via one-place buffers between components. Each component waits for inputs to arrive on *all* of its input buffers, then executes an enabled transition of its associated state machine, and finally writes to all of its output buffers; this scheme is further described in [HSE97]. The Synchronous execution model is supported by the validation and verification tool AutoFocus [Aut06]. This, however, results in "lock-step" executions of the components that have to be coordinated by an abundance of control messages on the communication medium.

To better cater to the reactive environment in the automotive domain we have developed the *Asynchronous* execution template. It implements an execution model where, upon receipt of a message, each component immediately executes an enabled transition and sends output on the appropriate ports. This model eliminates undesirable "waiting" as well as network flooding by control messages. We have developed a set of tools providing validation and verification of systems developed using the asynchronous execution model: one tool (S2Promela) is supporting formal verification and one (ServiceDebug) does testing. A third tool (MSCCheck) is currently work in progress; it will allow us to perform formal verification in a compositional fashion. The goal is to be able to verify also implementations and models that are too complex for the current tools.

Besides the *RT CORBA CodeGen*, we have developed *M2Aspects* to efficiently generate executable aspect-oriented implementations of service models [KMM06]. *M2Aspects* translates services into aspects; the aspect-weaving capabilities of the AspectJ compiler then create the executable implementation. We use multiple such prototypical implementations of the same service repository but with different target architectures for quick architecture exploration and validation [KMM06].

3.5 Tools for Verification and Validation

We have different options for validating M2Code models. AutoFocus [Aut06] is a good solution to address the synchronous model of execution. In the automotive domain, how-

ever, we usually prefer the asynchronous one. *S2Promela*, *ServiceDebug* and *MSCCheck* are our solutions for the asynchronous domain. All these tools work on the XML files generated by M2Code and provide different facilities to analyze the models.

ServiceDebug (Fig. 13) uses the code generation facilities described in the previous section and steps through the execution of each component via an interactive graphical interface. To support debugging of models, the tool shows a graphical representation of the state machines and their current states. The tool allows the user to choose the order each transition is taken by all state machines and even to inject messages on behalf of some component.

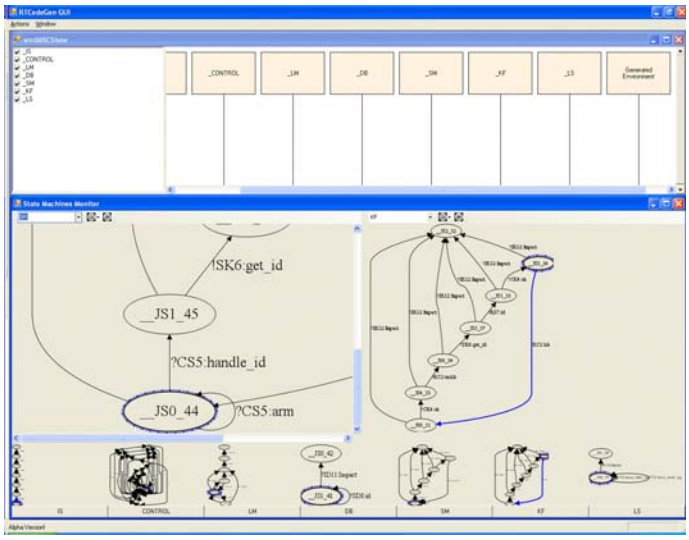


Figure 13: ServiceDebug User Interface

Another approach to validation applies formal verification techniques. To this end we have developed a prototypical tool able to translate the M2Code-generated component state machines to a suitable input file for a model checker. In particular we have chosen to generate a Promela source file that can be verified by the SPIN model checker [Hol03]. The tool is called S2Promela and is written in Java.

As described before, the service-oriented approach we apply is based on a rich ADL that provides two different description levels: a logical one, where the interactions between abstract roles are defined, and an implementation one, where logical entities are mapped to physical nodes of the target system. Our S2Promela tool uses models obtained from M2Code, by means of the XML interchange file, and additional information about the mapping of the service model to a deployment architecture, to generate a Promela description of the system.

Promela allows us to easily model concurrent programs communicating by message passing. We use the language facilities (*proctype* and *chan*) to map elements from our model to Promela constructs. In our translation each role is converted to a Promela *proctype* definition. The *proctype* parameters are *chan* variables representing the input/output channels

of the corresponding role. The translation of the state machine for each role is performed according to the schema suggested in [Hol03]. Each state is encoded by a label in the *proctype* and each transition by a guarded goto statement. Multiple enabled transitions are non-deterministically selected by using if-statements.

Each *proctype* defines a template for the behavior of a role. To activate each role and to allow it to perform a given function the *proctype* must be instantiated by means of a run-statement in the initialization (*init*) process. The process of initialization allows us to configure a system as required by the deployment model. In our CLS case study, for instance, there are two physical components playing the role IS (Fig. 4). This will be translated in two run-statements one for each instance of the role. Channels are instantiated and mapped accordingly to the deployment architecture.

The resulting Promela can be used to verify properties with the SPIN model checker. The properties to verify can be specified in all the ways accepted by SPIN. We have experimented with two ways of specifying properties. The first, is using LTL formulas. SPIN is able to convert those formulas to *never* claims which are included in the Promela file to perform the verification. This strategy for specifying properties is more indicated to specify constraints over the states of the automata generated by M2Code. For instance, we could express that the CONTROL Role mapped to Controller component will eventually reach state UNLD when one IS Role is in FINAL state. However, using LTL formulas is difficult in this setup to verify constraints on the messages exchanged on the various channels. For this reason we use another Promela construct: the *trace/notrace* command. Using these commands it is easy to express the interaction pattern to verify.

In our CLS example we have verified the property that when an *Impact* message is sent by an *ImpactSensor*, eventually an *unlck_ok* message is sent by *LockMotor*. The encoding of the property is:

```
notrace{  do  ::IC10!Impact;
                     do
                        ::IC10!Impact;
                        ::LC1!lck_ok;
                     od
              ::LC1!lck_ok;
              ::LC1!unlck_ok; od }
```

The *notrace* statement raises an error during verification if the sequence of messages is possible in any run. The fact that the verification did not return any error signifies that there exists no trace where the *Impact* message is sent and an *unlock_ok* does not follow. The SPIN model checker has explored and stored more than 11 million states to prove the correctness of our assertion and has used 1.7 gigabytes of RAM.

4 Experiences and Discussion

We have applied our service-oriented approach extensively to various projects, from the automotive domain to sensor networks to enterprise integration architectures [KMMP06].

We have used M2Code extensively in developing these projects. With the exception of the trigger operator, which is used for liveness specifications only, all of the operators mentioned above have been implemented. Our model-based approach also provides means to specify cross-cutting QoS properties of interaction-based scenarios on all levels of granularity, from an entire service down to a single interaction. In addition, it is a good platform for further tool integrations. We have successfully used the RT CORBA CodeGen tool to generate distributed prototypes and monitor real-time properties of case studies in the automotive domain.

We have used our S2Promela and ServiceDebug tools to verify critical properties of real time automotive domain projects. Because all tools make use of XML as the data exchange format, we can readily chain the mentioned tools, creating a tool chain from interaction definitions to code generation, runtime property monitoring, verification and model checking. Our service-oriented process is well-suited for building *fail safe* real-time systems, by concentrating on services as the main modeling elements. Failures are mostly cross-cutting; separation of logical and deployment models in our service-oriented process gives us the capability to lift failures from the deployment layer up to the logical layer, addressing them at the crosscutting service level. Building failure management into our process and tool chain is currently in progress.

5 Related Work

Triggered by its success in the telecommunications domain [Par02, ITU06] the term service has become quite prevalent in the literature, especially in the context of "web services" [STK02]. So far, however, services have been used mainly as an *implementation* concept, not as a first-class *modeling* entity. Consequently, existing definitions for the term service capture only syntactic lists of operations upon which a client can call. These definitions are inadequate for a systematic treatment of services throughout the development process. This is especially true also for the UML [OMG05] or SysML [Sys06], which do not recognize services as separate modeling entities. In our approach, the interaction-based service notion emerges as a cross-cutting modeling element regarding both system structure and behavior. In particular, we have established a decoupling between services (functions) as modeling elements and implementation infrastructures on top of which services can be implemented. We use a generalized notion of a system service in our interaction-based modeling approach. In [KNP04, KMM06] we present our service modeling approach based on the modeling of role interactions. It is related to the role concept introduced in [Pae97] and the activities of [KM96]. While our service concept is based on interaction patterns, stressing the cross-cutting nature of services, the roles of [Pae97] describe projections of such patterns onto individual components; to yield the overall picture the latter have to be recomposed into a global interaction specification. Activities of [KM96] capture global interaction properties as we do in our service definition; in contrast to our approach, however, [KM96] views activities as classes and roles as extensions to these classes.

The component-based development approach [Szy02] certainly has many advantages, including support for encapsulation, modularity, defining a unit of deployment, fault-containment, and many more. However, it falls short for cross-cutting aspects including interaction patterns. In contrast, by establishing a decoupling between service modeling and deployment of the resulting services on top of component architectures, we allow for "late binding" between functionality and components. Services provide a level of abstraction higher than components because services hide the components that implement them. This induces a choice regarding the architecture on top of which the services are implemented.

Our approach allows for the system to be understood at the granularity of individual features instead of components. The ability to gracefully deal with faults, both predictable and unpredictable, is an important property of embedded systems. Although we have not elaborated on this topic here, our approach allows for a better understanding of failures that emerge from the interplay of multiple components; the component-based approach accounts for faults localized to individual components.

Our approach is related to the Model-Driven Architecture (MDA) [OMG03] and architecture-centric software development (ACD) [OMG05]; similar to MDA and ACD we also separate the software architecture into abstract and concrete models. In contrast to MDA and ACD, however, we consider services and their defining interaction patterns as first-class modeling elements of *both* the abstract and the concrete models. Furthermore, we do not apply a transformation from abstract to concrete model. Our work is related to the work of Batory et al [SB98]; we also identify collaborations as important elements of system design and reuse. Our approach in particular makes use of MSCs as notation and is independent from any programming language constructs.

Architecture and tool support are the key instruments to address the complexity of real-time distributed systems. In [GH06], for instance, a framework is presented that allows replay of distributed real-time system based on architecture models. Our RT-CORBA CodeGen tool leverages similar model information to generate code for monitors that run in parallel to the system and inform the developer of unsatisfied deadlines.

Many graphical tools supporting software modeling have been researched and implemented. In [LMB$^+$01], for instance, the Generic Modeling Environment (GME) is presented, a tool aiming to support many modeling paradigms thanks to meta-model based configurations. M2Code, on the other hand, focuses on extended MSCs and HMSCs – features not currently supported by GME.

A tool suite particularly targeted for service engineering is the jABC environment [MS06] with its predecessor METAFrame [SM99]. Both offer behavior-oriented development, incremental formalization, and library-based consistency checking. The tool suite supports a service engineering development process and a modeling theory [MS06]. Synthesis is applied to generate executable prototypes which are abstracted in views, modified and verified until the behavior satisfies the requirements. The service notion we advocate in this paper generalizes the plugin-based service notion of jABC; in fact, as explained in the context of our Service-ADL, our approach to service-orientation brings forward cross-cutting

interaction aspects as elements of the logical architecture so that they can be implemented on a wide variety of deployment architectures.

6 Conclusions and Outlook

The high complexity of developing distributed, reactive systems in the embedded domain and other application domains requires effective development methodologies that mitigate these complexities. Complexities are often caused by the high degree of interactions in these systems. Model-based approaches promise to provide a solution to these challenges.

In this paper we have described a model-based approach to developing distributed, embedded systems. Our approach puts the concept of service in the focus of interest. Services are the first class elements that guide the development process from requirements analysis to deployment and execution of the realized system.

Tool support is essential to show the efficacy of a development approach and to increase the efficiency of its application in practical use. We have described how our methodology can be supported by means of tools. Our tool landscape contains tools for requirements analysis, service specification and architecture design, implementation and deployment and verification. The tools are connected in form of a tool chain. We have shown how our tool chain enables a software engineer to specify the basic reactive behavior of our case study example, the Central Locking System (CLS) and generate a distributed system prototype out of the specification model on top of a chosen deployment infrastructure and architecture configuration.

Future work will include a higher degree of integration between the tools and a seamless support of all model elements and operators through all phases of the development cycle.

7 Acknowledgments

Our work was partially supported by the UC Discovery Grant and the Industry-University Cooperative Research Program, as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2). Further funds were provided by the Deutsche Forschungsgemeinschaft (DFG) within the project *InServe*. We are grateful to the anonymous reviewers for insightful comments.

References

[AKMP05] Jaswinder Ahluwalia, Ingolf Krüger, Michael Meisinger, and Walter Phillips. Model-Based Run-Time Monitoring of End-to-End Deadlines. In *Proceedings of the Conference on Embedded Systems Software (EMSOFT)*, 2005.

[Aut06] AutoFocus. Website, 2006. `http://autofocus.informatik.tu-muenchen.de/index-e.html`.

[BKM06a] Manfred Broy, Ingolf Heiko Krüger, and Michael Meisinger, editors. *Automotive Software - Connected Services in Mobile Networks. Proceedings of the Automotive Software Workshop San Diego 2004*. Lecture Notes in Computer Science, Volume 4147, Springer, New York, 2006.

[BKM06b] Manfred Broy, Ingolf Heiko Krüger, and Michael Meisinger, editors. *Pre-Proceedings of the Automotive Software Workshop San Diego 2006*. UCSD, 2006. `http://aswsd.ucsd.edu/2006`.

[Bro05] Manfred Broy. The Impact of Models in Software Development. In *Lecture Notes in Computer Science, Volume 2605*, pages 396–406. Springer Verlag, 2005.

[Eva03] Eric Evans. *Domain Driven Design*. Addison-Wesley, 2003.

[GH06] Holger Giese and Stefan Henkler. Architecture-driven platform independent deterministic replay for distributed hard real-time systems. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 28–38, New York, NY, USA, 2006. ACM Press.

[Hol03] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[HSE97] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent Graphical Specification of Distributed Systems. In *FME'97*, volume 1313 of *LNCS*, pages 122–141, 1997.

[IT96] ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1996.

[ITU06] ITU. Sancho Definitions Database. Website, 2006. `http://www.itu.int/sancho`.

[KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[KM96] B.B. Kristensen and D.C.M. May. Activities: Abstractions for Collective Behavior. In *ECOOP'96*, volume 1098 of *LNCS*, pages 472–501. Springer Verlag, 1996.

[KM04] Ingolf Heiko Krüger and Reena Mathew. Systematic Development and Exploration of Service-Oriented Software Architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 177–187. IEEE, 2004.

[KMLS05] Ingolf H. Krüger, Reena Mathew, Stefan Leue, and Tarja Systä. Component Synthesis from Service Specifications. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 255–277. Springer Verlag, 2005.

[KMM06] Ingolf Krüger, Reena Mathew, and Michael Meisinger. Efficient Exploration of Service-Oriented Architectures using Aspects. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[KMMP06] Ingolf Krüger, Michael Meisinger, Massimiliano Menarini, and Stephen Pasco. Rapid Systems of Systems Integration - Combining an Architecture-Centric Approach with Enterprise Service Bus Infrastructure. In *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 51–56. IEEE, 2006.

[KNP04] Ingolf Krüger, Edward C. Nelson, and Venkatesh Prasad. Service-based Software Development for Automotive Applications. In *CONVERGENCE 2004*, 2004.

[Krü00] Ingolf Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[Krü03] Ingolf Heiko Krüger. Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In Mauro Pezzè, editor, *FASE 2003*, volume 2621 of *LNCS*, pages 387–402. Springer Verlag, 2003.

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. *Workshop on Intelligent Signal Processing, Budapest, Hungary, May*, 17, 2001.

[Mat04] Reena Mathew. Systematic Definition, Implementation and Evaluation of Service-Oriented Software. Master's thesis, University of California, San Diego, 2004.

[MS06] Tiziana Margaria and Bernhard Steffen. Service Engineering: Linking Business and IT. *Computer*, 39(10):45–55, 2006.

[NP03] Edward C Nelson and K V Prasaad. Automotive Infotronics: An emerging domain for Service-Based Architecture. In I. H. Krüger, B. Schätz, M. Broy, and H. Hussmann, editors, *SBSE'03 Service-Based Software Engineering, Proceedings of the FM2003 Workshop*, Technical Report TUM-I0315, pages 3–14. Technische Universität München, 2003.

[Obj02] Object Management Group: Real-time CORBA specification, 2002. `http://www.omg.org/technology/documents/index.htm`.

[OMG03] OMG (Object Management Group). Model Driven Architecture (MDA). MDA Guide 1.0.1, omg/03-06-01, 2003. `http://www.omg.org/mda`.

[OMG05] OMG (Object Management Group). UML, Version 2.0. OMG Specification formal/05-07-04 (superstructure) and formal/05-07-05 (infrastructure), 2005.

[Pae97] Barbara Paech. A Framework for Interaction Description with Roles. Technical Report TUM-I9731, Technische Universität München, 1997.

[Par02] Parlay 3.0, 2002. `http://www.parlay.org/en/specifications/`.

[Rus06] Yenny Rusli. Methodological Translation of Service-Oriented to Component-Oriented Specification. Master's thesis, University of California, San Diego, 2006.

[SB98] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of ECOOP 1998*, volume 1445 of *LNCS*, pages 550–570. Springer Verlag, 1998.

[SM99] Bernhard Steffen and Tiziana Margaria. METAFrame in Practice: Design of Intelligent Network Services. *LNCS*, 1710:390–415, 1999.

[STK02] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.

[Sys06] Systems Modeling Language (SysML), 2006. `http://www.sysml.org/`.

[Szy02] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[TRH+04] David Trowbridge, Ulrich Roxburgh, Gregor Hohpe, Dragos Manolescu, and E.G. Nadhan. *Integration Patterns. Patterns & Practices*. Microsoft Press, 2004.

[Zav01] Pamela Zave. Feature-Oriented Description, Formal Methods, and DFC. In *Proceedings of the FIREworks Workshop on Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2001.

From Component Models to Function Nets?

Jan Philipps

Validas AG
Garching b. München
philipps@validas.de

1 Introduction

By "model-based development" one frequently refers to the use of graphical descriptions instead of textual program code. The success of the Matlab/Simulink or ASCET tools, at least in the domain of control theory, shows that developers benefit from such graphical domain-specific languages.

Models are used not only as higher-level programming languages, however. Besides the obvious use for documentation, we can distinguish between two modes of use of models:

- In *constructive* mode, models are used as intermediary products that in some way or other will find their way into the system to be built. In constructive mode, models are either successive enriched by missing information (for instance, models containing structure information are augmented with additional behavioral information), or they are transformed (preferably automatically) into another intermediary product closer to the final system. During this transformation, usually additional information is added as well. Code generation is an example of such a transformation, where details about the concrete programming language, types and interfaces are added.
- In *analytical* mode one derives properties from a model, either as an aid to understanding of the system under construction, or as a reference for verification of later versions or artefacts. Such properties can serve as oracles for experiments on the final product—e.g., to describe the intended output of tests.

In this position paper, we discuss whether the modeling languages and heuristics for embedded software components are suitable for vehicle function nets as well.

2 Software Component Models

As mentioned above, modeling languages as graphical programming languages have become widely used for embedded systems including automotive systems. While this is in

particular true for software components dealing with continuous control, which are commonly described by block diagrams in Matlab/Simulink or ASCET, it is to a lesser degree also the case for components dealing with discrete control flow, which can be modeled in either of a number of state machine notations, for instance in the Statecharts of Rhapsody, or in the Simulink-extension Stateflow.

Code generated from such models, often extended with some wrapper code, can directly be incorporated as component code in electronic control units. Constructive use for software components is state of practice.

Analytic use, while still a topic of academic research, is also becoming successfully used:

- Some vendors offer verification tools to automatically verify certain properties of models (usually invariants);
- Test case generators produce sets of test sequences that allow one to test hand-written component code against the behavior described by a model; completeness of the set of test cases is usually determined by a coverage measure (of structural properties, or of a set of requirements).

3 Vehicle Network Models

It seems reasonable to lift software component models to the level of vehicle networks, where they are used to describe the network as a logical architecture or function net, in contrast with the technical architecture or physical network.

At the logical level, systems are described essentially as a network of *functions* that are interconnected by signals; the signals are typed according to the information they carry. In more ambitious approaches, the functions themselves are enriched with behavioral information, for instance by assigning a software component model to each function. In this way, one can even use the same modeling languages for both components and the logical architecture.

At the technical level, a vehicle network is described at the level of electronic control units (ECUs), networks and gateways.

The common development paradigm proposed is that vehicle network development starts with a logical architecture designed by the OEM, which is then mapped to a technical architecture. The supplier for each control unit integrates the various functions from the logical architecture that are mapped to their ECU.

The idea is that one may develop and reason about at least part of the vehicle network independent of the physical architecture. The major motivation here is firstly to obtain better intellectual control over the development process by separating purely functional issues from the technical issues, and secondly to achieve reuse for different technical architectures.

There are a number of problems with this idealized approach. Firstly, reuse and free deployment of functions is hampered in any situation where the responsibility for software

and control unit implementation is shifted to external suppliers. Secondly, the technical architecture has to satisfy a number of constraints, such as cost, size and placement of components and busses, that are closely related to the vehicle itself. Freedom of deployment of functions is only a secondary factor in the choice of the technical architecture. Consequently, disregarding reuse between different product lines, the technical architecture will be fixed earlier in development than the logical architecture. Thirdly, disregarding the real order of development of logical and technical architectures leads to a certain choice of abstractions used during modeling. It is, however, not clear that this choice is particularly fortuitous.

Below, these problems are considered for constructive and analytical modes of use of function net models.

4 Constructive Use

The decoupling of function nets from the underlying technical architecture leads to several problems:

- Infrastructure functionalities that are related to the ECUs are disregarded. This means that function nets fail to represent a large part of the complexity of automotive embedded systems, such as network management, self-tests, diagnostics, calibration, reprogramming or start-up and shut-down behavior.

  All of these aspects can be modeled in one language or another, and in software construction, some of these aspects might be dealt with by standardized middleware [HBS+06] or infrastructure libraries, but it is not clear how they could be integrated in a single model.

- Application functionalities must to some degree be able to cope with issues that arise at the technical level. Fault tolerance can perhaps be regarded as a separate issue at the software component level, but not at the vehicle network level.

  Unfortunately, the faults that have to be dealt with are highly dependent on the technical architecture; for instance, asynchronous (CAN) and synchronous (FleyRay, TTP) busses have different failure modes.

Of course, in spite of these difficulties function nets can serve as a valuable piece of documentation; it is not clear, however, whether this justifies the cost of constructing and maintaining them.

5 Analytical Use

The problems for analytical use of function nets are similar to those of constructive use: Without knowledge about the technical architecture, function nets are incomplete in that they disregard infrastructural functions, and the abstractions chosen restrict their use.

For instance, a widely-used form of analysis are reviews that examine a system model for the impact of component faults (e.g., FMEA or HAZOP). In principle, such analyses can be performed on function nets [MNP95]. But the failure modes examined by such an analysis are still dependent on the technical architecture. It is pointless to analyze a system under the wrong assumptions; it is too expensive to analyze it for all possible faults in all possible technical architectures.

6 Conclusion

Although embedded systems deeply depend on software, and although most innovations at least in embedded automotive systems depend on software, the final product still must be regarded as a physical object, and not only as a carrier of software.

This means that transferring established software engineering ideals to the embedded world is not without problems. Concepts such as that of a "logical" architecture capturing nominal behavior make sense in the pristine world of software engineering, where hardware may in some sense be regarded a commodity to be readily replaced, and where the faults of the underlying hardware are often disregarded.

The technical architecture of a vehicle, on the other hand, is carefully chosen to satisfy a number of electrical and mechanical and economical constraints that imply that it is the technical architecture that is designed first. Also, reliability and usability demands on a vehicle imply that vehicle functions must be designed in order to cope with problems at the technical level, something that is impossible without knowledge of that level.

Instead of an independently developed logical architecture that is to be mapped onto a technical architecture, it may instead be fruitful to strive towards models that incorporate knowledge of at least part of the technical architecture. Such models can be regarded as abstractions of the vehicle network in terms of services and interfaces, perhaps similar to the ideas presented in [RRJF03]. They could also be regarded as a separate representation of independent problems to be solved by the various vehicle functions, where each representation explicitly includes infrastructural functions like network management, diagnosis, calibration and reprogramming, and in particular their integration.

References

[HBS+06] H. Heinecke et. al. AUTOSAR – Current results and preparations for exploitation. In *7th EUROFORUM conference "Software in the vehicle*, 2006.

[MNP95] J. A. McDermid, M. Nicholson, and D. J. Pumfrey. Experience with the Application of HAZOP to Computer-Based Systems. In *Compass '95: 10th Annual Conference on Computer Assurance*, pages 37–48, 1995.

[RRJF03] A. Rae, P. Ramanan, D. Jackson, and J. Flanz. Critical feature analysis of a radiotherapy machine. In *International Conference of Computer Safety, Reliability and Security (SAFECOMP)*, 2003.

Obstacles to the Adoption of Model-based Design within the Automotive Supply Industry

Eva Kalix, Oliver Schuette

WABCO Development GmbH
Am Lindener Hafen 21
30453 Hannover, Germany
eva.kalix@wabco-auto.com
oliver.schuette@wabco-auto.com

Abstract: Model-based Design (MBD) is a well known concept in the automotive industry, but still has not gained widespread acceptance in the large software development departments of the automotive supply industry.

This article examines the reasons for this, looking at MBD from an economic point of view. An analysis is made to show where MBD may yield significant increase in productivity covering the specification, design and construction phase of embedded software. Three main topics are explored in the text: First, the role MBD plays in customer/supplier relations. Second, the framework that needs to be established to change to MBD and, in addition, the initial invest that comes with this change. Third, the reasons why the increase in productivity to be gained from automatic code generation is limited and subject to certain prerequisites.

1 Introduction

Within the automotive industry MBD is by now a well known concept. Many of the scientific challenges behind this technique have been met. During the last decade conferences with an industrial background focused on Hardware-In-the-Loop (HIL) systems. Nowadays, these are increasingly dominated by success stories regarding automatic code generation.

Despite these numerous success stories, model-based design has not gained widespread acceptance in the large software development departments of the automotive supply industry. Only HIL system testing, the last model-based development step in the V-model, is practically state of the art. Almost every supplier developing software possesses the knowledge necessary to not only test, but also create, embedded software using a MATLAB/SIMULINK based tool chain. However, it is done so only for a small portion of the software.

In this article we will explore the reasons for the cautious adoption of MBD as a general software development method.

Our basic assumption is that the driving force behind any technical progress is an increase in productivity. We will therefore go through the value chain of a software product analyzing in which phases MBD possibly yields significant increases in productivity.

We will reason that there are some phases where we do not expect MBD to gain ground, because the economic benefit is questionable. On the other hand there are phases where the gain in productivity is obvious, but technical limitations slow down the spread of this technology and many challenges concerning MBD remain [5], [6].

The point of view we are taking is that of an automotive supplier. We will therefore start from where a software product enters the realm of the supplier: A functional specification is agreed between the vehicle manufacturer (OEM) and the supplier.

2 OEM supplier relations within the automotive industry

The starting point is always an OEM ordering the realization of a functional specification from a supplier. The functional specification consists of a collection of requirements defined in a document, a database (e.g. DOORS) or possibly in UML in future cases. Additionally it may also be accompanied by functional simulation models ("executable specification") [4]. Literature about MBD often emphasizes the advantages of using simulation models as means of specifying requirements [7], [8]. The benefit is seen in the continuous use of the same model from the specification to the final implementation phase. The model is merely augmented by additional information during each step.

To see the catch in this argumentation it is necessary to return from an engineering to a purely economic point of view and to ask for the supplier's economic raison d'être. There are two main reasons why the overall efficiency increases, if a product (in our case an ECU consisting of software and hardware) is manufactured by a supplier. On the one hand there are efficiency gains due to the supplier's higher degree of specialization. But on the other hand even more important are the efficiency gains due to economies of scale.

A supplier usually serves all OEMs with a set of similar electronic devices from the same source code base. The supplier receives a number of functional specifications from different OEMs, filters these for overlapping requirements and consolidates all customer specifications to one product specification. As it brings down unit costs this consolidation is usually of advantage to all parties involved.

Bearing in mind that the main task of a supplier is to consolidate the functional specifications of various customers, it is now possible to return to the previous discussion of the role MBD plays in the requirements phase. A requirement defined, for example, in a graphical SIMULINK model, is on the one hand far more exact than a verbal specification. On the other hand we lack the methodology to find out whether two block diagrams describe roughly the same requirement. These two characteristics of a SIMULINK model obstruct the continuous use of a model all the way from OEM to ECU. To employ MBD in the early specification phase does, to a certain extent, contradict the objective of reducing unit costs.

Despite this, there are examples of very successful cooperation where a supplier has directly implemented the OEM's functional model. Naturally, this is always the case

where unit costs are not the central focus. OEMs are increasingly using software features as unique selling points for their products and specifically compete on those features. In this case the core attribute is the uniqueness. The OEM is specially interested that the supplier is not able to consolidate requirements. Here a specification via models is an effective way of enforcing this. This particular type of business definitely exists, but accounts only for a small part of the overall business activity [2].

The central argument of this section is that the use of executable, functional models in the requirements phase mainly depends on the business case behind it. It is somewhat dysfunctional where economies of scale are the main focus, but very effective where differentiation and uniqueness is the central goal.

3 MBD in Functional Design and Test

The next phase we are looking at is the functional design phase after the product specification has been consolidated.

In this phase economic reasons to employ MBD are easy to find. The earlier an error is discovered during the development process, the cheaper it can be revised. In conventional development, tests are done after the implementation of a module. One of the great advantages of model-based design is that most of these tests are shifted to a functional development phase before the implementation starts.

Model-In-the-Loop (MIL) simulations as well as rapid control prototyping are classic examples. The functional design in MIL mode is a cost-effective and rapid way to validate and test control algorithms independently from and before their implementation.

Considering the many advantages that can be gained from the early testing possibilities of MBD the central question of this section is: Why do we still find software development methods other than MBD?

A key issue is certainly the high initial investment that has to be made in order to change the development method. Besides, the high license costs, the tool chain, the process chain, and the employee skills need a major update.

First of all, it is a necessary precondition of efficient MBD that an environment model exists which can interact plausibly with the control algorithm that is to be designed, e.g. a dynamic vehicle model.

Furthermore, a layered software architecture is needed where the functional applications are not mixed with hardware dependent device drivers. Interfaces of software modules must be clearly defined, and data that are passed between the modules must be encapsulated [9]. Then it is easily possible to cut out an application module and design it model-based. In contrast, hardware related modules of the basic layers can not be tested at all by MIL. Therefore, they should be coded manually.

The benefit of software layers and encapsulated modules have been acknowledged and is enforced by initiatives like AUTOSAR. However, most of the already existing software does not follow this modern approach and MIL strategies for design and test cannot be used effectively or cannot be used at all.

The development of environment models for MIL itself can be very time consuming. Whereas the cheapest model is the one that has not been developed, but reused, as in the case of production code that is (re)used for different software versions. Usually one model for each type of product is enough. Variant management can be used with the environment model to adapt model parts to different software variants and revisions, e.g. to different vehicle types.

Complexity of a model depends on its purpose. In the case that it is set up for control design, a model must "only" copy the controller relevant dynamic behaviour of the environment outside the ECU. In case an overall simulation is aspired to the environment model can be very detailed. Its purpose is, for example, to test hardware diagnosis features, robustness concerning tolerance of electronic circuits or signal bus behaviour. An overall simulation helps the system designer to validate system functionality, whereas the developer usually picks only those model parts relevant for software packages on the to-do list.

The highest benefit can be achieved if simulation and design tools of all development domains interact together neatly. For example, the software is prototyped on hardware models that are linked dynamically to the current CAD design.

In contrast to MIL scenarios, Hardware-In-the-Loop (HIL) simulation is state of the art, despite the fact that it is also a test procedure based on environment models. Tests are performed using a series production ECU, on which the production code is running. However, model and ECU software can be developed totally independently except for the similar ECU interface.

HIL is post design, post implementation testing. Although the real time environment needed for an HIL test bench is expensive the gain of productivity of an HIL system obviously is high, because neither vehicle nor driver is needed to perform a test. Some vehicle states can be adjusted more easily in simulation. Special failure reactions, e.g. on cable breaks, can be triggered in simulation only. Tests can be repeated in exactly the same way for each new software revision.

Introducing MIL sets requirements regarding software architecture and the software development process. Using only HIL does not. When focusing on the overall system test, an HIL environment model can usually be less detailed than a MIL model, i.e. it is much simpler to obtain and much cheaper regarding human power [3]. And, an HIL test bench frees human resources, whereas a detailed MIL model binds them in the first instance.

As already stated above, the optimal performance of model-based design is achieved when models can be reused to a great extend. Therefore, it makes sense to build a model suitable for HIL testing as well as MIL prototyping.

Last but not least a sophisticated model can close the gap between SW and HW development domains and can ease the communication between developer teams. If models are setup strategically for company wide use, the model the SW developer uses for functional prototyping maybe (part of) the same model that the HW developers runs for parameter studies. This concurrent MBD has the power to further decrease time to market. However, CMBD requires a very stable process of file exchange and communication for the regularly needed model updates.

The central statement of this section regarding the paper's key issue is that high initial investments in software architecture and environment models slow down the adoption of

MBD in functional design. The reason why HIL simulations could become such an immediate and sweeping success is that this technology does not require any alterations in software architecture and runs with less elaborate environment models, thus lowering the initial invest considerably.

4 MBD during implementation

From the early 80s on, software engineers tried to attribute software development cost to the different development stages [1]. An ever-present result of all these investigations is that the costs of implementation are small compared to the overall costs. Therefore, it is quite surprising, that discussions regarding model-based design focus increasingly on automated code generation.

Surely the attractiveness of autocode generation is strongly related to the popularity of SIMULINK. This tool has been made available to most current graduates. However, at universities, SIMULINK is usually only used for the design of floating-point algorithms that were comfortably converted to C by the RealTime-Workshop. Fixed-point applications are outside the academic sphere. It is evident that a SIMULINK trained engineer, who has to produce fixed-point C code, welcomes an easy-to-use autocode generator. Mechanical engineers, for example, can be skilled and fast in functional prototyping with a graphical tool, but are often inexperienced, slow and inefficient with regard to C. The fixed-point autocode generator equips them with C abilities but not necessarily also with competence concerning data types, scalings and code efficiency. Fixed-point information in the models, as well as the code itself, has to be programmed, rsp. checked carefully by an advanced C programmer.

If a perfectly tested and functionally optimized model-based algorithm is available, it becomes an obvious step to use it as a base for the ECU implementation of the software. For experienced programmers it is less work to configure the autocode generator correctly than implement the module by hand. In contrast to hand coding, autocoding tools offer an extremely helpful direct comparison of floating-point and fixed-point results. Overflow warnings for fixed-point data types are also issued. Problems introduced by fixed-point error can easily be debugged.

However, provided that a SIMULINK model needs to be created just for autocoding of a detailed specified algorithm and test scenario, no gain of productivity can be seen when compared to hand coding. This is particularly true, if a typical industry development scenario is taken into account: Many developers work on one project, which has many development branches. The MATLAB/SIMULINK tool chain is currently hardly capable of handling such a development process.

The central argument of this section is that automatic code generation by itself is not likely to lead to major gains in productivity. This is simply because coding only accounts for a small portion of the software development costs. One way it can be effective is as a spin-off of an already established model-based functional design process. This means a tested functional model exists anyway.

5 Summary and conclusion

The discussions above support the following conclusions:

"Executable specifications" are not likely to alter the manufacturer supplier relations on a wider scale in the near future. This is mainly because the intrinsic redefinition of the suppliers' role does not have a strong economic incentive.

Post specification MBD has a strong economic foundation, but requires a high effort to meet the prerequisites for its introduction. Any measure capable of reducing this initial investment would greatly facilitate the spread of MBD.

Automatic code generation can thrive only in a mature and well established model-based development process. It should therefore be the last, and not the first, process to be introduced.

References

[1] Boehm, B.W. and Papaccio, P.N.: *Understanding and Controlling Software Costs*, IEEE Transactions on SW Eng., Vol. 14 (10), 10/1988.

[2] Humphrey, R.: *Model Based Development for Automotive Body Systems.* IAC 2005, Dearborn, The MathWorks.

[3] Lebert, K.Pfister, F.; Schantl, R.;Beer, W.: *Productive use of hardware-in-the-loop systems for the calibration process*; Proceedings of the 1. Int. Symposium für Entwicklungsmethodik, Wiebaden 10/2005

[4] Prabhu, S.M. and Mosterman, P.J.: *Modelling, Simulating, and Validating a Power Window System Using a Model-Based Design Approach*. Automotive Digest, The MathWorks, 11/2003

[5] Rau, A.: *Model-Based Development of Embedded Automotive Control Systems*. Berlin: dissertation.de - Verlag im Internet GmbH, 2003; Zugl.: Tübingen, Univ., Diss., 2002. ISBN: 3-89825-599-9.

[6] Rau, A.: *Potential and Challenges for Model-based Development in the Automotive Industry*. In "Business Briefing: Global Automotive Manufacturing and Technology", World Market Research Center, 10/2000.

[7] H. Schlingloff et al.: IMMOS – *Eine integrierte Methodik zur modellbasierten Steuergeräteentwicklung*. Forschungsoffensive des BMBF "Software Engineering 2006", Berlin.

[8] Schlingloff, H. et al.: *Modellbasierte Steuergerätesoftwareentwicklung für den Automobilbereich*. In: GI-Tagung "Automotive Safety and Security 2004 - Sicherheit u. Zuverlässigkeit für autom. Informationstechn.″, Stuttgart, 10/2004.

[9] Schuette, O. and Kalix, E.: *Standardized SW Modules for Model Based Design Workflow*. International Automotive Conference, 5/2006, Stuttgart, The MathWorks.

Herausforderungen bei der Neugestaltung von Seriensoftwareentwicklungsumgebungen

Andy Yap, Peter Großhans

Model Based Engineering
MB-technology GmbH
Kolumbusstrasse 2
71063 Sindelfingen
andy.yap@mbtech-group.com
peter.grosshans@mbtech-group.com

Abstract: Die Entwicklung von Seriensoftware für Automotive-Steuergeräte beinhaltet die Berücksichtigung zahlreicher Begleitprozesse, die vom Kunden (OEM bzw. 1st Tier Supplier) vorgeschrieben sind. Hierbei ist oft auch die Verwendung bestimmter Tools vorgesehen, die in die vorhandene Toolkette zu integrieren sind. Außerdem muss die Toolkette bestimmte Ein- und Ausgabeformate unterstützen. Anpassungen an der Toolkette, z. B. bei einem Versionswechsel der eingesetzten Tools, sind immer noch häufig mit einem erheblichen Aufwand verbunden.

1 Einleitung

Dieser Beitrag stellt einige Aspekte der Anwendung von modellbasierter Entwicklung für Embedded Automotive Anwendungen vor. Er gibt einen Einblick in die produktdomänenspezifischen Anforderungen an die Software in Kapitel 2, die Prozesse in Kapitel 3 und die Herausforderungen bei der Veränderung von Entwicklungsumgebungen in Kapitel 4.

2 Anforderungen an Entwicklungsergebnisse

Bei der modellbasierten Entwicklung von Seriensoftware für Embedded Automotive-Systeme sind neben den allgemein gültigen Anforderungen (mehr bei [Ra02] bzw. [KC04]) an Software-Entwicklungsumgebungen folgende Punkte aus dem Produktkontext besonders zu beachten.

1. Variablenskalierung für die Zielarchitektur: Die Entwicklungsumgebung muss eine unterschiedliche Variablenskalierung für verschiedene Zielplattformen (mehr bei [HY04]) ermöglichen. Die Komplexität heutiger Modelle (zum Teil über 20000 Blöcke) erfordert eine Unterstützung durch Data Dictionarys.

2. Variantenmanagement: Unter dem Begriff Variantenmanagement verstehen wir hier die Verwaltung unterschiedlicher Parameterwerte, die z.B. für mehrere Fahrzeugvarianten erforderlich sind (z. B. Rechts-/Linkslenkerfahrzeug). Je Variante sind verschiedene Werte pro Parameter nötig. Somit sieht der Parametercode (data.c/data.h in Abb.1) für jede Variante unterschiedlich aus.

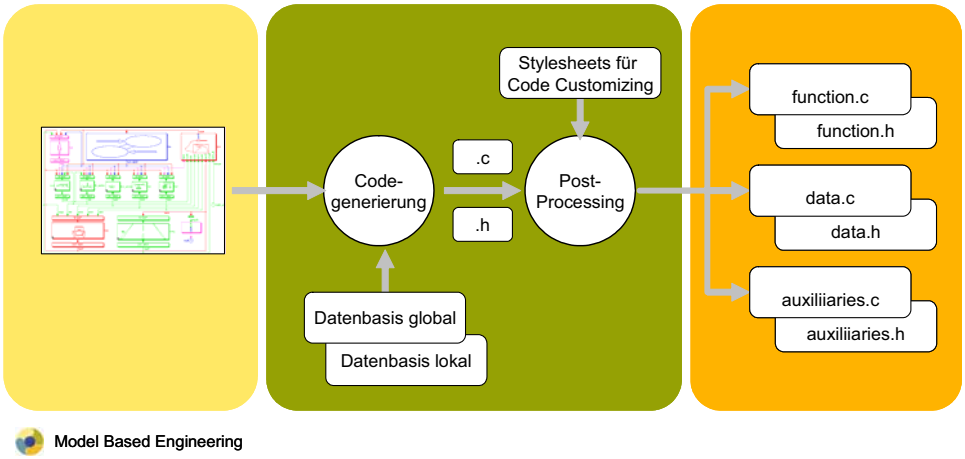


Abbildung 1: Übersicht über den Ablauf einer Codegenerierung

3. Messtechnik: Jede Art von Messsystem erfordert entsprechende Dateiformate (z.B. ASAP Files). Dabei müssen Messtechnikdaten in Datenstrukturen liegen. Oft sind dazu auch bestimmte Formatierungen (Stylesheets in Abb.1) für die generierten Code-Dateien vorgegeben.

4. Kalibrierung: Unter Kalibrierung wird hier die Änderung von Parameterwerten der Funktion im Steuergerät während der Entwicklungsphase verstanden. Es gibt Online- bzw. Offline-Konzepte für die Kalibrierung. Bei einer Online-Kalibrierung ist das Steuergerät im normalen Betriebszustand, eine Verstellung von Werten findet zunächst im RAM statt und kann später im nichtflüchtigen Speicher abgelegt werden. Bei der Offline-Kalibrierung wird das Steuergerät beispielsweise in einem speziellen Modus gebootet, der über das Kalibrierungstool die Daten in den nichtflüchtigen Speicher schreibt. Wie bei der Messtechnik muss die Entwicklungsumgebung bestimmte Ausgabedatenformate liefern können.

5. Diagnose: Automotive Steuergeräte müssen heute schon während der Entwicklungsphase zahlreiche Diagnosedienste unterstützen. Für den Fall, dass Parameter auch über Diagnosedienste geändert werden sollen, muss klar sein, wie das erwartete Format der entsprechenden Daten ist, z. B., wenn die Parameterwerte in Datenstrukturen abgelegt sind, werden den entsprechenden Diagnosediensten Zugriffsmakros für die Parameterverstellung zur Verfügung gestellt.

6. Versionsmanagement für Modelle und Tools: Die Verwaltung der Modelle und Modellumgebungen für verschiedene Versionen, Baureihen usw. ist bei der heutigen Komplexität nur noch mit Toolunterstützung (CVS, PVCS, SVN o.ä.) zu leisten. Auch die Entwicklungsumgebung muss in das Versionsmanagement eingebunden sein.

Neben diesen Anforderungen an die Entwicklungsumgebung sind auch die Tools rund um die Codegenerierung zu berücksichtigen. In Abb. 1 ist dargestellt, wie der Prozess der Codegenerierung gegliedert ist. Zuerst ist die Funktion des Codegenerators selbst dargestellt, der die Umwandlung der grafischen Notation in die Hochsprache umfasst. Anschließend erfolgt das Post-Processing, in welchem kundenspezifische Anforderungen an den generierten Code umgesetzt werden, wie die Aufteilung von Programmcode und Daten in verschiedene Files oder die Generierung zusätzlicher Header-Files für den Export bestimmter Daten. Je nach verwendetem Codegenerator sind diese Features „built-in“ und es müssen z.B. nur Stylesheets konfiguriert werden, oder es sind hierzu zusätzliche Tools oder Skripte notwendig.

3 Konzepte und Prozesse für den Einsatz der Entwicklungsumgebung

Die immer komplexeren Funktionen in den Steuergeräten sind heute nur mit mehreren Entwicklern zu bewältigen, die jeweils eine Teilfunktion bearbeiten. Für die verteilte Entwicklung gibt es verschiedene Konzepte für die Entwicklung und die Integration. Jedes der Konzepte weist Vor- und Nachteile auf, die im Zusammenhang mit dem im Projekt geltenden Entwicklungsprozess bewertet werden sollten. Die Konzeptionen und ihre Vorteile werden weiter unten in diesem Kapitel vorgestellt.

Fall 1: An jedem Arbeitsplatz ist eine vollständige Entwicklungsumgebung mit einem simulationsfähigen Gesamtmodell installiert. In dieses Gesamtmodell ist das Modell der Teilfunktion eingebunden, welches der Entwickler[1] bearbeitet. Die vollständigen Datenbeschreibungen liegen in einer lokalen Datenbasis. Zu festgelegten Zeitpunkten wird von einem Integrator[2] ein Abgleich der lokalen Datenbasen durchgeführt und die einzelnen Teilmodelle in ein neues Gesamtmodell integriert.

---

[1] Aufgaben des Entwicklers: Der Entwickler hat die Aufgabe die Anforderungen an seine Funktion termingerecht und mit hoher Qualität umzusetzen. Er dokumentiert seine Entwicklung in allen Schritten (Pflichtenheft, Kommentare in Modell/Code, Testprotokolle) und nimmt an Reviews seiner Arbeiten teil. Für die Validierung seiner Funktion erstellt er Testfälle und führt Funktionstests in einer geeigneten Testumgebung aus.

[2] Aufgaben des Integrators: Er muss sicherstellen, dass die Arbeitsprodukte (Funktionen) der beteiligten Entwickler zu den in der Releaseplanung vereinbarten Zeitpunkten vorliegen und in das Gesamtsystem integrierbar sind. Er stellt je nach Projektkontext (Fall 1-5) den Entwicklern eine Integrationsumgebung zur Verfügung, in der sie ihre Funktionen entwickeln. Er ist verantwortlich für die Bereitstellung der Integrationsstände an den Kunden. Er muss sehr gute Kenntnisse bezüglich der Funktion und der Konfiguration der eingesetzten Toolkette haben, ebenso von den spezifischen Besonderheiten der Zielplattformen.

Dieses Vorgehen hat den Vorteil, dass alle Entwickler bei der Entwicklung ihrer Funktion selbstständig arbeiten können. Dies ist dann möglich, wenn sich die Schnittstellen der Funktionen im System selten ändern und die Entwicklungszyklen zeitlich so gestreckt sind, dass die Integrationsphase entsprechend lang ausfallen kann, da hier in der Regel Nacharbeiten an Funktionen notwendig sind, z.B. wegen Namenskonflikten oder geändertem Laufzeitverhalten von Signalen.

Fall 2: An jedem Arbeitsplatz ist ein Teil der Entwicklungsumgebung für die Bearbeitung der jeweiligen Teilfunktion installiert. Ein individuelles Umgebungsmodell ermöglicht den Test der Teilfunktion. Die Datenbasis beinhaltet nur die im Modell der Teilfunktion verwendeten Daten. Zu festgelegten Zeitpunkten werden die einzelnen Modelle zu einem Gesamtmodell und die Datenbasen zu einer Gesamtdatenbasis integriert.

Dieses Vorgehen hat den Vorteil, dass die Entwickler von den anderen Funktionen bzw. vom Gesamtsystem nur die Schnittstelle kennen müssen. Dies ist dann möglich, wenn die einzelnen Funktionen des Gesamtsystems weitgehend rückwirkungsfrei voneinander sind. Der Integrationsaufwand ist bei diesem Vorgehen mindestens so groß wie in Fall 1.

Fall 3: An jedem Arbeitsplatz wird auf die zentrale Datenbasis zugegriffen. Die versionierten Modelle der einzelnen Teilfunktionen liegen auf einem Server. Der Entwickler arbeitet immer mit dem aktuellen, versionierten Gesamtstand. Eine Integration wird von der zentralen Datenbasis durchgeführt.

Hier liegt der Vorteil darin, dass der Aufwand bei der Integration kleiner wird, weil ein Teil dieses Aufwands von der Toolkette übernommen wird. Dies stellt höhere Ansprüche an die Toolkette in Bezug auf die Vernetzbarkeit und setzt bei den großen Datenmengen komplexerer Systeme ein leistungsfähiges Netzwerk voraus.

Fall 4: Hier wird wie im Fall 2 vorgegangen, es wird aber von jedem Modell der Teilfunktionen der generierte Code abgeliefert. Es gibt kein (simulationsfähiges) Gesamtmodell, stattdessen werden die generierten Codes der Teilfunktionen integriert.

Dieses Vorgehen kann dann erforderlich sein, wenn das Gesamtsystem so groß und komplex ist, dass es mit den heute zur Verfügung stehenden Tools zur Codegenerierung nicht mehr (sinnvoll) verwaltbar ist. Ein anderes Szenario für dieses Vorgehen kann sein, dass die Funktionsmodellierung beim OEM, die Integration aber beim Lieferanten erfolgt, wo eventuell auch noch weitere Funktionen in das System integriert werden, die mit einem anderen Prozess entwickelt wurden. Hierbei geht natürlich ein wesentlicher Aspekt der modellbasierten Entwicklung verloren. Dies sollte natürlich durch umso intensivere, modellbasierte Tests der Funktionen und eine Wiederverwendbarkeit der Testfälle ausgeglichen werden.

Fall 5: Hier wird wie im Fall 3 vorgegangen. Es wird an jedem Arbeitsplatz mit einer Read-only-Kopie der zentralen Datenbasis gearbeitet. Die Daten der Teilfunktionen werden in einer separaten Datenbasis verwaltet. Bei der Integration werden die einzelnen Datenbasen der Teilfunktionen zur Gesamtdatenbasis integriert.

Jeder der vorgestellten Fälle hat entsprechend auch Auswirkungen auf die Toolkette in Bezug auf Lizenzbedarfe und Art der Installationen, die auf den verschiedenen Arbeitsplätzen vorhanden sein müssen.

4 Herausforderungen beim Versionswechsel

Bei einem Versionswechsel eines oder mehrerer Tools in der Toolkette muss vor der Umstellung eine Planung erfolgen, die die Abfolge der Schritte festlegt und eine mit allen Beteiligten abgestimmte Roadmap für die Migration beinhaltet. Zusätzlich ist die Migration vorab an einem Testprojekt zu überprüfen.

Folgende Fragestellungen sind im Vorfeld zu klären:

- Welche Features werden benötigt bzw. für welche Zielplattformen muss Code in welcher Form generierbar sein?

- Welche Features werden bislang von welchem Tool in der Toolkette abgebildet?

  Erfolgt beispielsweise. die ASAP-Generierung durch den Codegenerator oder durch ein separates Tool.

- Welche Features können künftig von neuen Tools in der Toolkette übernommen werden?

  Ist beispielsweise die komplette Formatierung des Codes künftig mittels Stylesheets möglich, möglicherweise ist dann kein externes Tool mehr erforderlich.

- Welche Features wurden bislang über ein Post-Processing des generierten Codes umgesetzt und an welcher dieser Stellen kann künftig darauf verzichtet werden?

  Anmerkung: Dies ist bei einer (Tool-)Strategie sinnvoll, die darauf setzt, Konfigurationsdaten möglichst zentral zu halten.

- Erfordert die Umstellung der Toolkette eine Änderung der bisherigen Prozesse?

  Eventuell ist der Arbeitsablauf zu ändern, siehe oben beschriebene Fälle.

Vorgeschlagen wird die Einrichtung eines Migrationsteams. Dieses Team erstellt die Roadmap und trifft sich in regelmäßigen Abständen, um den Status von Spezifikation, Implementierung und Roll-out mittels geeigneten qualitätssichernden Maßnahmen zu überprüfen.

5 Zusammenfassung

In den ersten beiden Kapiteln wurde die Vielfalt der möglichen Ansätze deutlich. Je nach Art des Projektes und den beteiligten Unternehmen an der Entwicklung wird eine individuelle Gestaltung des gesamten Entwicklungsprozesses nötig. Daraus ergeben sich viele Variationen der Ausgestaltung. Dazu kommen noch unterschiedliche Innovationszyklen bei den eingesetzten Elementen der Entwicklungsumgebungen und der Bedarf einer vergleichsweise extrem langen Laufzeit der Projekte, die sich an den Produktzyklen der Automobilindustrie orientiert. Daraus wird der Aufwand für eine Umstellung deutlich, die im letzten Abschnitt behandelt wird. Als Ausblick darf durch den Einfluss von Standardisierungen auf eine einfachere Vorgehensweise gehofft werden.

Literaturverzeichnis

[HY04] Grantley Hodge, Jian Ye and Walt Stuart: Multi-Target Modelling for Embedded Software Development for Automotive Applications, 2004 SAE World Congress, Detroit, Michigan, March 8-11, 2004

[KC04] Klein, T., Conrad, M., Fey, I., Grochtmann, M.: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. Lecture Notes in Informatics (LNI), Vol. P-45, Köllen Verlag, 2004, pp. 31-41

[Ra02] Rau, Andreas: Model-Based Development of Embedded Automotive Control Systems. Dissertation, Universität Tübingen, 2002

Using Simulink® and Real-Time Workshop® Embedded Coder for Safety-Critical Automotive Applications

Mirko Conrad

The MathWorks GmbH
Adalperostr. 45
85737 Ismaning
mirko.conrad@mathworks.de

Abstract: When developing safety-critical embedded software it is important to consider the objectives of standards such as IEC 61508. These standards impose additional constraints on the development processes and require the production of evidence that the objectives were met.
Unfortunately, IEC 61508 is a generic safety publication not targeted at the specifics of in-vehicle applications. Its lifecycle model does not fit the automotive development processes. Furthermore, the standard was developed for traditional (hand-coded) software development processes and does not cover advanced automotive software development technologies such as Model-Based Design and code generation.
Therefore the objectives of IEC 61508 need to be interpreted for automotive applications and the measures and techniques recommended by the standard must be mapped onto Model-Based Design processes and tools.
This paper discusses the usage of Model-based Design tools from The MathWorks for safety-critical automotive applications and outlines a solution to facilitate the demonstration of compliance with the IEC 61508 objectives.

1 Model-Based Design of Automotive Applications and IEC 61508

Model-Based Design is becoming the preferred software engineering paradigm for the development of embedded controls in central vehicle subsystems, such as powertrain and chassis. The main advantages of this approach are that software development time is reduced and product innovation is enhanced through the use of executable specifications and production code generation. These technologies are successfully used to produce software for safety-critical applications, see e.g. [Pot04, TMW05], but extra consideration is needed to address the constraints imposed by standards and to produce the required evidence for compliance demonstration.

Due to the lack of sector specific standards, the generic safety publication IEC 61508 IEC 61508] is increasingly considered to be relevant for safety-critical automotive applications. This standard with the somehow bulky title 'Functional safety of electrical / electronic / programmable electronic safety-related systems' is a sector-independent safety publication for electrical / electronic / programmable electronic safety-related systems (short: E/E/PES).

The standard was developed originally by the process and automation industries. Its seven parts (referenced as IEC 61508-1 to IEC 61508-7) were published between 1998 and 2000, i.e. in the early Model-Based Design era. IEC 61508-3 is concerned with the requirements for software development. Due to its origin in non automotive industries the lifecycle model of IEC 61508 does not fit very well for automotive applications [Pof04], however upcoming standards are seeking to address this [ISO 26262].

IEC 61508 can be considered as a prescriptive standard, which provides detailed lists of techniques and measures with recommendations. The amount of techniques / measures depends on the safety integrity level (SIL) of the application. To aid the selection of software safety techniques and measures appropriate to the required safety integrity IEC 61508 provides so called software safety integrity tables ranking the various techniques/measures against the four safety integrity levels SIL 1 to SIL 4 (see annex A of IEC 61508-3). As an example IEC 61508-3 highly recommends the usage of language subsets for SIL 3 and 4 applications (cf. IEC 61508-3, Table A.3). The related extract of the safety integrity table is given in Table 1.

Table 1: Example of a software safety integrity table as provided by IEC 61508-3 (extract)

A.3: Software design and development: Support tools and programming language

Technique / Measure	SIL1	SIL2	SIL3	SIL4
...	...	...	...	...
3 Language subset	o	o	++	++
...	...	...	...	...

++ ... The technique or measure is highly recommended for this SIL.
+ ... The technique or measure is recommended for this SIL as a lower recommendation to a ++ recommendation.
o ... The technique or measure has no recommendation for or against being used.

Altogether, annexes A and B of IEC 61508-3 include 19 software safety integrity tables recommending 120+ measures and techniques (see the appendix of this paper for an overview). Cross-referenced to the software safety integrity tables is an overview of techniques and measures in annex C of IEC 61508-7.

These tables are intended to be tailored for a particular safety critical application, i.e. for each recommended or highly recommended technique/measure it has to be documented which measures were implemented and / or which replacement measures were employed. As a rule, if a particular highly recommended technique or measure is not used then the rationale behind not using it should be documented and agreed with the safety assessor. The tailoring process for the software safety integrity tables is illustrated

by means of worked examples in annex E of IEC 61508-6. The tailored software safety integrity tables are used within the compliance demonstration process and are necessary for system certification. They are used as evidence that the objectives of the standard were met.

The tailoring is a lengthy and labor intensive process but the tailored tables for similar projects typically contain common parts. According to [CD06, CD06a] the project-specific accruing effort for the tailoring can be reduced effectively by factoring out these common parts. This can be realized if respective adapted tables covering typical automotive development projects and typical tool chains are made available.

Since IEC 61508 was written with traditional (hand-coded) software development processes in mind the standard does not cover advanced automotive software development technologies such as Model-Based Design and code generation. Therefore, the recommended techniques and measures are targeted at hand-coded development procedures and need to be mapped onto Model-Based Design processes and tools making the tailoring more complex. For example, the recommendation for language subsets in Table A.3 leaves open if this is related to the modeling language, e.g. Simulink®, or the implementation language, e.g. C, or both.

Compliance Demonstration

In order to ensure a consistent interpretation of the standard across different Model-Based Design projects and to avoid unnecessary multiple work in projects comparable to each other, adapted tables covering typical Model-Based Design workflows and industry standard tools such as the Simulink [SL] and Real-Time Workshop® Embedded Coder [RTW-EC] can be used.

The project-specific accruing effort for this can be reduced effectively if respective generic versions (a.k.a. templates) of these tables for typical Model-Based Design projects are made available. These templates are then instantiated project-specifically.

In order to create the template tables, the original software safety integrity tables from the standard are extended by an additional column labeled ‘Applicable Tools / Processes for Model-Based Design’. The entries in the column describe how a particular measure or technique can be supported by products or solutions for Model-Based Design. Typically the applicable Tools / Processes for Model-Based Design are split up into model level and code level activities.

Table 2 shows an extract of an annotated IEC61508-3 table which corresponds to the example in Table 1. The rationale behind the entries of the additional column is as follows.

In Model-Based Design language subsets occur on the design level (a.k.a., modeling language subset) as well as on the code level (a.k.a., implementation language subset). Pure subsetting of the modeling and/or the implementation language is inadequate in many cases Furthermore, compliance to an implementation language subset shouldn't be enforced only after the code has been generated. Instead it is important to have modeling and code generation guidelines that restrict the use of blocks and suggest proper code generation settings. Making recommended patterns and style guides available has also proven useful.

Table 2: Example of a generic, annotated software safety integrity table (extract)

A.3: Software design and development: Support tools and programming language

Technique / Measure	SIL1	SIL2	SIL3	SIL4	Applicable Tools / Processes for Model-Based Design
…	…	…	…	…	…
3 Language subset	o	o	++	++	The MAAB Style Guides and/or organization specific modeling guidelines can be used to define a subset of the modeling language. The Simulink Block Data Type Support section of the Simulink documentation lists the blocks, which can be used for code generation with Real-Time Workshop Embedded Coder. MISRA-C and/or organization specific coding guidelines can be used to define a subset of the implementation language. Restricted language subsets can be partially enforced by using Simulink Model Advisor Restricted modeling language subsets can be enforced by model reviews based on reports generated by Simulink Report Generator. Restricted implementation language subsets can be enforced by code reviews based on Real-Time Workshop Embedded Coder – Code Generation Reports. Third-Party Products such as PolySpace Desktop can be used to verify MISRA-C: 2004 compliance. Third-Party Products such as PolySpace Desktop can be used to check language subset considerations within the generated code.
…	…	…	…	…	…

According to [Bär05], MISRA-C compliance is an implementation of a coding standard as required by IEC 61508-3. TÜV Süd accepts MISRA-C [MISRA04] if adherence to the MISRA-C rules can be shown and the compliance to the coding guidelines will be verified by static code analysis tool. The OEM Initiative Software (HIS) also recommends to use the entire set of MISRA-C:2004 guidelines. For well-founded exceptional cases deviations are to be documented following the deviation procedure described in section 4.3.2 of the actual MISRA-C document [HIS06].

To address MISRA-C compliance, MathWorks maintains a MISRA-C compliance analysis package for Real-Time Workshop Embedded Coder since Release 12.1. This

package is based on model inspections, code analysis, and quality engineering product tests suites. The MISRA-C compliance package includes an overview document, a compliance matrix, and presentation of violations and mitigations. Simulink and Stateflow model examples as well as Real-Time Workshop Embedded Coder code examples are also included.

Some believe that coding standards for production code generation and hand coding may differ partially. Even MISRA launched an activity in 2005 to address the specifics of using visual modeling languages and automatic code generation for automotive systems. However, MathWorks MISRA compliance package is based on the full MISRA-C guidelines. The full guidelines are important to consider since hand code is often integrated with automatically generated code for production ECUs.

Simulink is a general purpose visual modeling language. It can be used to create controller as well as plant models. So not all modeling features and settings provided, are appropriate for modeling automotive control software. The MAAB style guidelines [MAAB01] were developed to address this issue. They became a popular source of information for the derivation of company specific modeling language subsets, pattern and guidelines. For safety-critical applications additional patterns and guidelines might be useful.

In summary, the MISRA-C and MAAB guidelines are means to partially fulfill the IEC 61508-3 requirements on language subsets and coding standards. To verify guidelines compliance, powerful verification and validation capabilities are essential.

Based on the generic templates in Table 2 the project specific effort can be reduced to the documentation of the deviations from the generic version. In particular it is necessary to document which measures were implemented and/or which replacement measures were employed. The following steps apply:

1. Selection of the SIL level for the application under development and removal of the columns for the non-applicable SIL levels.
2. Selection of a set of measures / techniques for the particular application (in case of alternate or equivalent techniques/measures).
3. Renaming of the column ‘Applicable Tools / Processes for Model-Based Design’ into ‘Interpretation in this application’.
4. Labeling of the measures / techniques for the particular application depending on whether they are ‘used’, ‘used to a limited degree’ or ‘not used’. If a particular measure / technique is not used, a reason has to be provided and the Applicable Tools / Processes for Model-Based Design belonging to it need to be removed.
5. Application-specific modification/refinement of the remaining annotations in the column ‘Interpretation in this application’

Table 3 exemplifies the results of the customization process for the language subset part of Table 2.

The derived application-specific tables are supposed to be submitted to the certification authority as part of the compliance demonstration process. It is recommendable to submit a first version of the tables early in the life cycle in order to be discussed and agreed with the assessor.

Note, that compliance with the IEC 61508 standard does not ensure the safety of the software or the application. But it can be demonstrated that state-of-the-art procedures were applied.

Table 3: Example of a derived application-specific table (extract)

A.3: Software design and development: Support tools and programming language

Technique / Measure	SIL3	Interpretation in this application
...	...	...
3 Language subset	++	Used: The MAAB Style Guides are used as subset of the modeling language. MISRA-C:2004 is used as subset of the implementation language. The modeling language subset is as far as possible enforced by using Simulink Model Advisor. All reported deviations and Modeling Language considerations which cannot be checked automatically need to be manually reviewed. PolySpace Desktop is used to verify MISRA-C: 2004 compliance. All reported deviations need to be manually reviewed.
...	...	...

Summary and Conclusion

Embedded automotive applications are increasingly used for safety-related or safety-critical applications. Most of these applications are based on software. Accordingly, engineers require embedded software development tools and processes that include proven, state-of-the-art quality assurance measures. A means to meet this requirement is to base the in-vehicle software development process upon the IEC 61508 safety standard.

IEC 61508 requires systematic software engineering processes and additional quality assurance measures in order to facilitate product safety and to minimize the remaining risk.

Model-Based Design with code generation can be used with IEC 61508. An appropriate combination of different products from the Simulink product family addresses a broad spectrum of the IEC 61508-3 objectives related to software. MathWorks' products including Simulink and Real-Time Workshop Embedded Coder have been successfully deployed for safety-critical applications up to the highest criticality levels.

However, since IEC 61508 dates back to the early Model-Based Design area and does not addresses popular automotive development technologies such as production code generation, it has been subject to interpretation. Mapping onto automotive development practices using Model-Based Design can be a time-consuming process.

In the past, the mapping of the IEC 61508 objectives onto Model-Based Design processes and tools was done on a per project basis. Significant portions of this mapping were caused by bridging the gap between the standard and modern automotive software development approaches and not by the specifics of the project under consideration. This resulted in unnecessary high efforts for the individual projects.

This paper proposed a more standardized way to facilitate and document IEC 61508 conformant Model-Based Design processes. First, the software safety integrity tables of IEC 61508 were enhanced in a way that they provide a project independent mapping of recommended techniques / measures onto tools of the Simulink product family and related processes. These template tables can then be easily tailored according to the needs of a particular project. The application-specific tables resulting from the tailoring can be used as evidence within the compliance demonstration process as suggested by IEC 61508-6.

Providing annotated versions of the software safety integrity tables covering typical Model-Based Design workflows is a powerful means to offer guidance for Model-Based Design projects using the Simulink family of products that must meet the IEC 61508 objectives.

In addition, using this approach cuts down the project specific efforts for compliance demonstration and reduces time-to-system-certification.

The outlined approach is not restricted to IEC 61508, is also applicable to the upcoming automotive safety standard ISO 26262 [ISO 26262, FP06].

Please contact the author for further discussion.

References

[Bär05] Andreas Bärwald: IEC 61508 & MISRA C - The Benefits of Utilising IEC 61508 and MISRA C for Automotive Applications. 1st IEE Automotive Electronics Conference, London, UK, 2005

[CD06] Mirko Conrad, Heiko Dörr: Deployment of Model-based Software Development in Safety-related Applications - Challenges and Solutions Scenarios. Proc. Modellierung 2006, Innsbruck, Austria, 2006, LNI Vol P-82, pp. 245-254

[CD06a] Mirko Conrad, Heiko Dörr: Einsatz von Modell-basierten Entwicklungstechniken in sicherheitsrelevanten Anwendungen: Herausforderungen und Lösungsansätze. Proc. Dagstuhl-Workshop 06022 Modellbasierte Entwicklung eingebetteter Systeme II (MBEES'06), Schloß Dagstuhl, Germany, 2006, pp. 1-17

[FP06] Matthias Findeis, Ilona Pabst: Functional Safety in the Automotive Industry, Process and methods. VDA Winter meeting, 2006

[HIS06] HIS Working Group Software Test: Gemeinsames Subset der MISRA C Guidelines. Version 2.0, 2006

[IEC 61508] IEC 61508-3:1998. International Standard IEC 61508 Functional safety of electrical/electronic/ programmable electronic safety-related systems. 1st edition, 1998

[ISO 26262] ISO 26262:2005. Road vehicles - Functional safety: Working Draft, 2005

[MAAB01] MathWorks Automotive Advisory Board: Controller Style Guidelines for Production Intent Development Using MATLAB, Simulink, and Stateflow. Version 1.00, 2001

[MISRA04] MISRA-C:2004. The Motor Industry Software Reliability Association: Guidelines for the use of the C language in critical systems. 2004

[Pof05] Ekkehard Pofahl: The application of IEC 61508 in the automotive industry. 2005

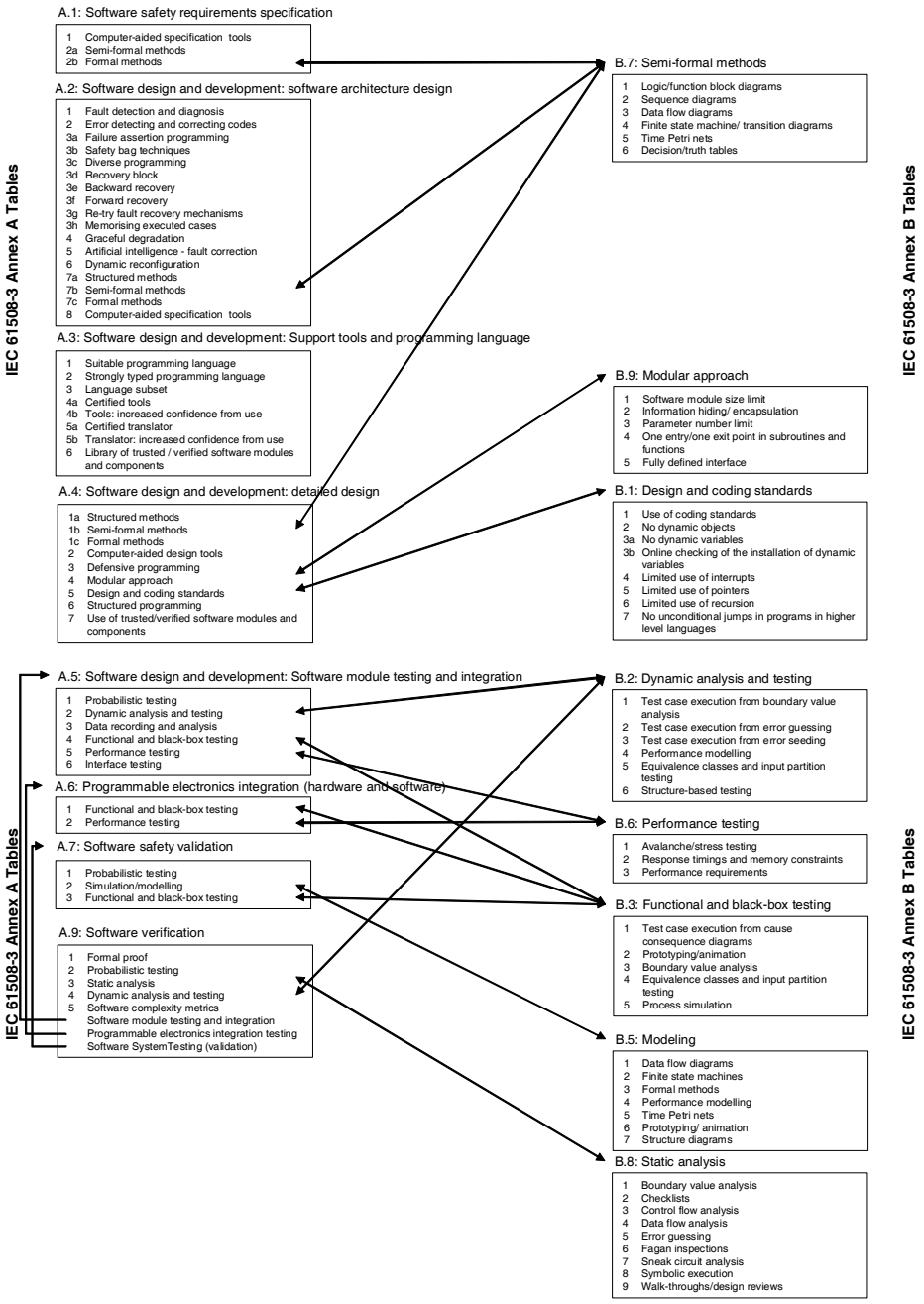
[Pot04] Bill Potter: Use of The MathWorks Tool Suite to Develop DO-178B Certified Code. ERAU / FAA Software Tools Forum, Daytona Beach, FL., US, 2004

[RTW-EC] Real-Time Workshop® Embedded Coder User's Guide. Version 4, The MathWorks Inc, 2006

[SL] Using Simulink®. Version 6, The MathWorks Inc, 2006

[TMW05] Alstom Generates Production Code for Safety-Critical Power Converter Control Systems, User Story, The MathWorks, 2005
www.mathworks.com/products/rtwembedded/userstories.html?file=10591

Appendix: Overview of IEC 61508-3 Software Safety Integrity Tables



A.8: Modification

1 Impact analysis
2 Reverify changed software module
3 Reverify affected software modules
4 Revalidate complete system
5 Software configuration management
6 Data recording and analysis

A.10: Functional safety assessment

1 Checklists
2 Decision/truth tables
3 Software complexity metrics
4 Failure analysis
5 Common cause failure analysis of diverse software (if diverse software is actually used)
6 Reliability block diagram

B.4: Failure analysis

1 Test case execution from boundary value analysis
2 Test case execution from error guessing
3 Test case execution from error seeding
4 Performance modelling
5 Equivalence classes and input partition testing
6 Structure-based testing

Notation und Verfahren zur automatischen Überprüfung von temporalen Signalabhängigkeiten und -merkmalen für modellbasiert entwickelte Software

Carsten Gips, Hans-Werner Wiesbrock
IT Power Consultants*†

Abstract: In diesem Artikel wird ein neuer Ansatz vorgestellt, Signalverläufe zu beschreiben und -abhängigkeiten zu erfassen. Anstelle einer konstruktiven Beschreibung, d.h. ausgehend von dem zeitlichen Verlauf und Stützpunkten, wird hier vom Werteverlauf ausgegangen. Merkmale und Eigenschaften der Signale werden deskriptiv erfasst und führen ihrerseits nun zu einer geeigneten Partitionierung, die dann auch algorithmisch überprüft werden kann. Diese Abstraktion dient im weiteren als Ausgangspunkt zur Formulierung und algorithmischen Auswertung von Signalabhängigkeiten.

1 Einleitung

Die Testauswertung in der modellbasierten Entwicklung eingebetteter Systeme ist in der Regel sehr zeit- und personalaufwändig und, manuell durchgeführt, selber hochgradig fehleranfällig. Die Bewertung langer Versuchsläufe und zahlreicher Signaldaten ist mühselig; kleine, aber signifikant fehlerhafte Abweichungen können auf Grund geringer Bildschirm- oder Druckauflösung leicht übersehen werden und auch Tagesform des Prüfers, Überlastung oder Ablenkung nehmen Einfluss auf sie.

Eine Standardisierung der anzuwendenden Kriterien sowie ihre Operationalisierung ist hier erstrebenswert und Voraussetzung einer weiteren Automatisierung.

Für reine Softwaretests existieren bereits erprobte Algorithmen und Verfahren für den Signalvergleich, die in dem Tool *MEval* ([mev]) umgesetzt wurden. Doch diese Algorithmen und Kriterien sind nicht unmittelbar auf Hardware- oder Systemtests anwendbar, da sie Referenzdaten voraussetzen. Als herausfallendes Beispiel seien hier die wichtigen Fahrversuche im Automotivebereich genannt. Die Fahrereingaben liegen meist nur in qualitativ beschriebener Form vor und sind so nur begrenzt reproduzierbar. Messungenauigkeiten und Zeitverzögerungen sind weitere Faktoren, die einen einfachen Signalvergleich nutzlos machen.

In Abb. 1 sind zwei fiktive Fahrverläufe dargestellt, die Messergebnisse eines Fahrversuches und einer entsprechenden Simulation eines Modells beschreiben könnten. Unter der

*Tel: +49 - (0)30 - 46 79 98 43, Web: `www.itpower.de`

†Diese Arbeit wurde durch das Land Berlin und die EU im Rahmen des Förderprogramms ProFIT der Investitionsbank Berlin unter der Antragsnummer 10127540 gefördert.

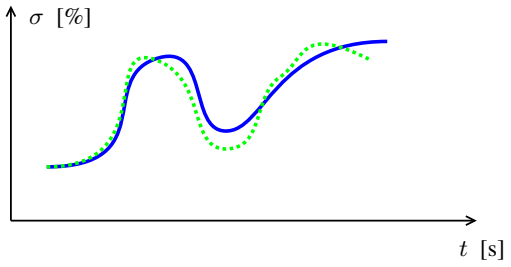


Abbildung 1: Beispiel Signalverläufe

Annahme, dass es sich bei den Signalen um eine Bremsverstärkung handelt, könnte man Abb. 1 folgendermaßen interpretieren: Zunächst bremst der Fahrer stark. Danach wird für eine kurze Zeit der Druck auf das Bremspedal zurückgenommen und anschließend wieder (diesmal weniger stark als bei der ersten Bremsung) gesteigert. Beide Kurven ähneln sich in ihrer „Gestalt“ bzw. ihrer Form. Um diese Aussagen aber überprüfbar zu machen, muss der intuitive Begriff der „Gestalt“ operationalisiert, formalisiert werden. Dann kann auch der Entwickler an einem anderen Standort dieses Verhalten nachvollziehen und dahinterliegende Anforderungen werden testbar.

Aber nicht nur für den Test ist eine solche Formalisierung erstrebenswert. Zu Beginn der Entwicklung sind die geforderten Signalergebnisse in der Regel nur in ihrem groben Verlauf, d.h. qualitativ, bekannt. Diese Anforderungen an den Signalverlauf müssen geeignet beschrieben werden, um das Verständnis des Entwicklers für die Implementierung zu verbessern und die Qualität zu erhöhen. Auch hier gilt es, eine geeignete Formalisierung zu wählen, die vom Spezifikator, dem Entwickler und Tester gleichermaßen verstanden werden kann und operational geeignet ist, automatisiert ausgewertet zu werden.

2 Der Ansatz

Um algorithmisch auswertbar zu sein, aber auch zur Förderung des Entwicklerverständnisses von erwarteten Signalverläufen ist eine Beschreibung durch einige wenige Kenngrößen angebracht. Eine Operationalisierung legt ebenfalls die Charakterisierung durch endlich viele Daten nahe.

Zur Beschreibung von Eingangssignalen eines Systems gibt es bereits zahlreiche Ansätze. Es bietet sich hier zunächst an, die Verläufe konstruktiv durch Rampen, Splines etc. und endlich viele Stützpunkte zu beschreiben. Viele erprobte Testsysteme setzen hier auf (CTE, MTest: [GG93, Con04]). Als weiterreichender Ansatz wäre hier auch die Zerlegung in Zustände zu nennen (TPT, [Leh03]). Jedoch reichen all diese Ansätze nicht allzu weit, will man die Verläufe der Ausgangssignale ebenso erfassen. Fragt man Spezifikatoren und Tester nach den erwarteten oder richtigen Verläufen, so hört man zunächst von ihnen: »Erst steigt das Signal stark an bis ein Extremum erreicht wird, dann bleibt es eine Weile

konstant ...«. In ihrer Beschreibung gehen sie meist nicht primär von einer zeitlichen Beschreibung aus, sondern von bestimmten Merkmalen und ihren Reihenfolgen, Dauern und auch Abhängigkeiten.

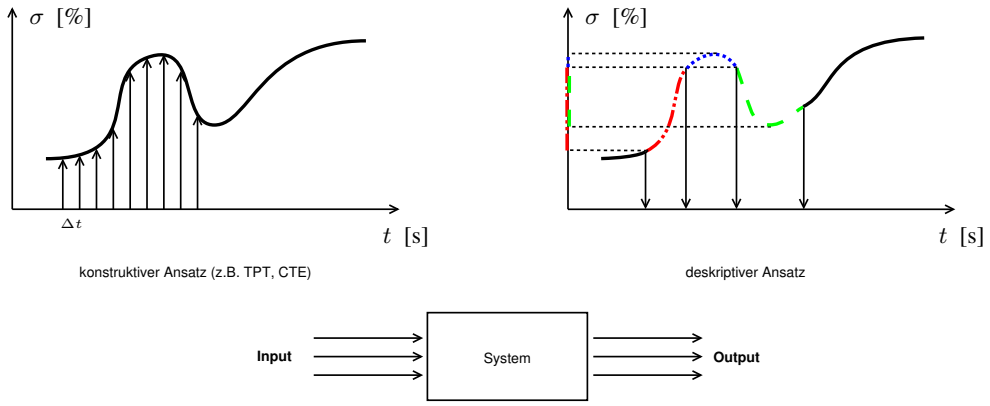


Abbildung 2: Überblick zum deskriptiven Ansatz

Der Ansatz, der in diesem Artikel vorgestellt werden soll, geht deshalb von Signalmerkmalen und -eigenschaften aus und entwickelt daraus eine Charakterisierung bzw. Diskretisierung der Signale (vgl. Abb. 3). Die geforderten Signalverläufe, d.h. die zeitliche Abfolge bestimmter Signalmerkmale mit entsprechenden Ausprägungen, werden zunächst über einfache Diagramme qualitativ beschrieben. Die Auswertung einer solchen Spezifikation erfolgt dann algorithmisch und liefert als Resultat eine endliche Zeitreihe, an denen die verschiedenen definierten Eigenschaften erfüllt sind. Auf diese Weise wird der konkrete Signalverlauf zu einer endlichen Charakterisierung abstrahiert. Ausgehend von dieser Diskretisierung lassen sich dann auch allgemein Abhängigkeiten zwischen verschiedenen Signalen spezifizieren. Diese Abhängigkeiten können dann in einem folgenden Schritt wiederum algorithmisch überprüft werden. In den Kapiteln 3 und 4 skizzieren wir die zugrunde liegenden Algorithmen.

Die Autoren sind der Ansicht, dass dieser Merkmals- oder Eigenschaft-bezogene Ansatz zur Beschreibung der ausgangsseitigen Signale geeignet ist und weiter reicht als ein konstruktiver, der zu sehr von einer Zeiteinteilung geleitet wird. Ähnliche Ansätze finden sich auch in [ZNSP06]. Da die Signalverläufe reproduzierbare Ausgaben von Steuergeräten sind, müssen sie deterministisch beschrieben und überprüft werden. Neuronale Netze und andere gängige Verfahren aus dem Bereich der Künstlichen Intelligenz, die im weitesten Sinne auf statistisches Lernen oder Auswerten zurückgreifen, sind deshalb leider nicht sehr geeignet.

Die hier vorgestellten Algorithmen und Darstellungen weisen eine gewisse Ähnlichkeit zu Verfahren aus dem Bereich Model-Checking (z.B. [AHLP00, GHK$^+$06]) auf. Im Gegensatz zu einem endlichen Automaten oder gewissen Kontrollstrukturen geht es hier aber um die Beurteilung und Auswertung von kontinuierlichen Signalverläufen. Ihre qualitative Beschreibung durch Diagramme ähnelt den geläufigen Zustandsdiagrammen, doch

dienen diese hier zur endlichen Charakterisierung und Abstraktion der kontinuierlichen Daten. Sie haben so keinerlei Selbstzweck und ihre Untersuchung ist sekundär. Nach der Abstraktion der Signalverläufe können wir jedoch temporallogische Forderungen an die endlichen Datensätze stellen, beispielsweise ob das Merkmal P_1 des Signalverlaufes von σ_1 vor dem Eintreffen des Merkmales P_2 von σ_2 stattfand etc. Deshalb lassen sich ein Teil der beim Model-Checking gemachten Erkenntnisse auch hier nutzen, jedoch ist der Kontext deutlich verschieden.

An einem einfachen Beispiel soll dieser Ansatz erläutert werden. Anstelle der Zeiteinteilung »für 5 s« oder »für eine Dauer von etwa 20 s« werden hier zunächst die Werteverläufe analysiert: »steiler Anstieg«, »Extremum«, »Abfall«, ... Erst anschließend werden die dazugehörigen Zeitintervalle betrachtet. Es erfolgt also eine Einteilung der Signale zunächst auf der „Wertachse“, die dann eine entsprechende Diskretisierung bzw. Charakterisierung auf der Zeitachse nach sich zieht.

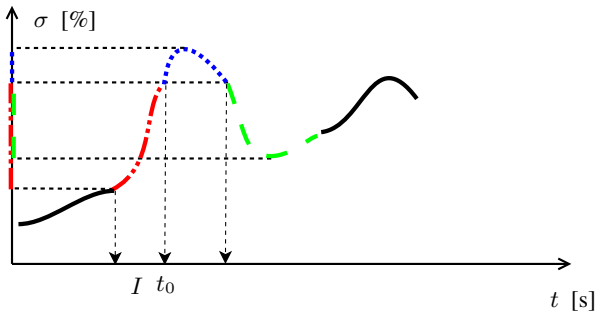


Abbildung 3: Charakterisierung der Signale anhand von Merkmalen

Im Folgenden sollen einige typische Merkmale informativ eingeführt werden, die in einem Signalverlauf leicht identifizierbar sind. Dazu zählen Extrema, Bereiche steilen Abfalls oder Anstiegs, spezielle Werte wie Nulldurchgänge oder Flag-Änderungen (vgl. Abb. 4). Aber auch das Frequenzspektrum kann ein relevantes Merkmal von Signalen sein.

Neben ihrer „Gestalt“ sind aber auch gegenseitige Abhängigkeiten wichtige Merkmale von Signalen! Aufsetzend auf der oben eingeführten Charakterisierung können auch Abhängigkeiten nun systematisch erfasst werden, wie in Abschnitt 4 näher ausgeführt wird.

Sei ein Signal durch seine Merkmale (Eigenschaften) und einer dazugehörigen Diskretisierung beschrieben worden. Dann kann man das Messsignal auf die Erfüllung dieser Merkmale abgleichen. Ebenso lassen sich zeitliche Abhängigkeiten zwischen Signalen jetzt betrachten: Ein Merkmal m_1 des Signales σ_1 darf erst auftreten, wenn *vorher* Signal σ_2 das Merkmal m_2 aufgewiesen hat. So bildet die Charakterisierung auch die Basis, um Signalabhängigkeiten zu formulieren.

Wie eine automatisierte Testauswertung auf solchen Beschreibungen aufsetzen kann, soll in den folgenden Abschnitten skizziert werden. Im ersten Teil (Abschnitt 3) dieser Arbeit

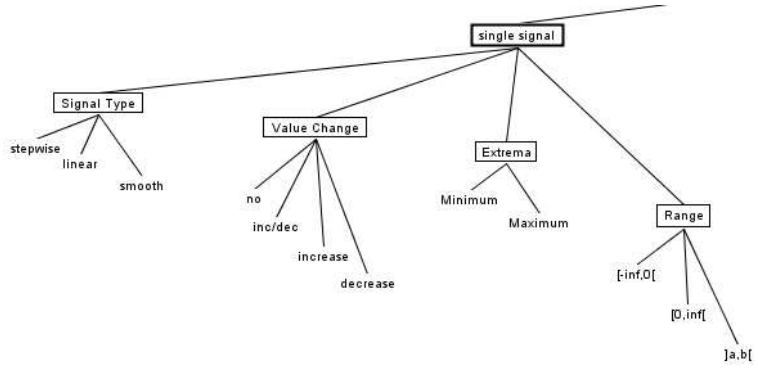


Abbildung 4: Übersicht wichtiger Signalmerkmale

wird das Konzept einer Notation vorgestellt, mit deren Hilfe diese qualitativen Beschreibungen von Einzelsignalen formalisiert werden kann. Im zweiten Teil (Abschnitt 4) wird ein Formalismus eingeführt, der Abhängigkeiten zwischen Signalen fassen kann und es werden Algorithmen skizziert, die, basierend auf der vorangehenden Notation, eine automatische Auswertung erlauben.

3 Signale und Eigenschaften

Signale sind Abbildungen, die zu jedem Zeitpunkt, an dem das Signal definiert ist, diesem einen (Signal-) Wert zuordnen:

Definition 3.1 (Signalverlauf)
Ein Signalverlauf ist eine Abbildung

$$\sigma : D_\sigma \to \mathbb{R}$$

mit $D_\sigma \subset [0, \infty]$.

(Diese Definition lässt sich leicht auf allgemeinere Signale mit weniger strukturierter Zeit und/oder Wertebereich verallgemeinern. Um die wesentlichen Ideen zu skizzieren, soll hier nur der einfachste Fall betrachtet werden.)

Eine Eigenschaft eines Signals ist eine Abbildung, die für ein Signal zu jedem Zeitpunkt, an dem das Signal definiert ist und diese Eigenschaft erfüllt ist, ein *true* (1) liefert, ansonsten *false* (0) ist:

Definition 3.2 (Eigenschaft (Property))
Eigenschaften sind Abbildungen

$$P : \{\mathbb{R}^{\mathbb{R}}, \mathbb{R}\} \to \{0, 1\}$$

Die Erfüllungsmenge $\mathfrak{M}_{P,\sigma}$ einer Eigenschaft P zu einem Signal σ ist definiert durch

$$\mathfrak{M}_{P,\sigma} := \{t \in \mathbb{R} \mid P(\sigma, t) = 1\} \cup \{\infty\}$$

mit $t \notin D_\sigma \Rightarrow P(\sigma, t) := 0$.

Eigenschaften von Signalen gelten also in bestimmten Zeitbereichen bzw. -punkten. Diese Zeitmengen sind ihre Erfüllungsmengen.

Ein Verlaufsdiagramm zur Beschreibung von Einzelsignalen ist nun ein bipartiter Graph mit eindeutigem Start- und Endknoten. Man unterscheidet also zwei Knotenarten: Phasen und Events. Diese sind durch gerichtete Kanten, den Transitionen, verbunden.

Events sind zeitpunktartige Ereignisse, die selbst keine Dauer haben. Events werden in dieser Arbeit mit dargestellt. Typische Beispiele sind: Extrema, Nulldurchgänge, etc.

Phasen sind Eigenschaften, die über eine bestimmte, spezifizierbare Dauer erfüllt sind. Sie haben eine Mindestdauer von einer Schrittweite (z.B. ein Taktschritt) und Anfangs- und Endzeitpunkt. Phasen werden im Folgenden durch graphisch notiert. Als Beispiel mag dienen: »Steiler Anstieg > 5, Wert in [2,5] . . . «.

Definition 3.3 (Erfüllungsmenge der Events)
Sei σ ein Signal und E ein Event mit Eigenschaft P. Dann ist die Erfüllungsmenge des Events definiert durch

$$\mathfrak{M}_{E,\sigma} := \{t \in \mathfrak{M}_{P,\sigma}\} \cup \{\infty\}.$$

Definition 3.4 (Erfüllungsmenge der Phase)
Sei σ ein Signal und Φ eine Phase mit Eigenschaft P und Mindest- bzw. Maximaldauer $[d_1, d_2[$. Dann ist die Erfüllungsmenge der Phase definiert durch

$$\mathfrak{M}_{\Phi_{[d_1,d_2[},\sigma} := \{t \in \mathfrak{M}_{P,\sigma} | [t, t + d_1] \subset \mathfrak{M}_{P,\sigma} \text{ und } [t, t + d_2] \not\subset \mathfrak{M}_{P,\sigma}\} \cup \{\infty\}$$

Start- und Endknoten sind spezielle Signalelemente. Sie haben genau eine ausgehende und eine eingehende Transition zu einer Phase. Der Startknoten hat den Zeitpunkt 0, für den Endpunkt ist dieser Zeitpunkt definiert durch den maximalen Wert des Definitionsbereiches. In jedem Pfad müssen Phasen und Events bzw. Switches alternieren (siehe Abb. 5).

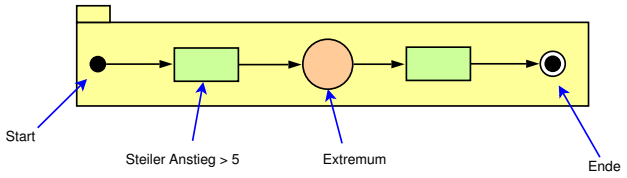


Abbildung 5: Verlaufsdiagramm zur Beschreibung eines einfachen Signalverlaufes

Switches sind besondere Events mit mehreren Eingangs- und Ausgangstransitionen. Es soll in dieser Übersicht aber nicht weiter auf diese Elemente eingegangen werden. Einige

Symbol	Element	Bedeutung
	Startevent	Kennzeichnet den Startzeitpunkt des Signales
	Endevent	Bezeichnet den Endzeitpunkt
	Event	Allgemeines zeitpunktartiges Ereignis
	Phasen	Dauerhafte Eigenschaft, besitzt eindeutigen Eintritts- und Ausgangszeitpunkt. Mindestdauer: 1 Schrittweite
	Switch	Erlaubt alternative Verlaufsformen, spezielles Event.
	Transition	Beschreibt die zeitliche Abfolge von Events, Phasen, etc.
[2, 5]	Temporale Implikation	Spezifiziert temporale Abhängigkeiten zwischen Ereignissen und Phasen
[1, 2]	Temporale Eigenschaft	Charakterisierung eines Ereignisses durch temporale Bezüge zu einem anderen
	Signal	Spezifikation für ein Signal
	Modul	Zusammenfassung von mehreren Elementen in ein parametrierbares Template

Tabelle 1: Elemente zur grafischen Beschreibung von Signalverläufen

weitere Elemente in Verlaufsdiagrammen zur grafischen Beschreibung von Signaleigenschaften sind in Tab. 1 aufgeführt.

Mittels solcher Verlaufsdiagramme lassen sich nun Signalverläufe qualitativ beschreiben: »Zunächst ist wichtig, dass das Signal einen steilen Anstieg von mindestens 5 hat (erste Ableitung hat den Wert 5). Anschließend wird ein Extremum erreicht, ...«.

Dieses Diagramm ist aber nun formal genug, um eine operationalisierte Auswertung zu erlauben. Sei ein beliebiges Signal gegeben, dann werden für jedes Element des Diagrammes die Zeitbereiche bestimmt, in denen das Signal diese Eigenschaft erfüllt, also die Erfüllungsmengen bestimmt. Genauer, es werden die frühest möglichen Erfüllungszeitpunkte (und bei Phasen auch die frühest möglichen Endzeitpunkte) beginnend mit dem Startknoten sukkzessive bestimmt. Erfüllt dieses Signal eine im Verlaufsdiagramm angegebene Eigenschaft nicht, so wird ihr frühester Erfüllungszeitpunkt auf ∞ gesetzt und das Signal erfüllt nicht das Diagramm.

Definition 3.5 (frühest möglicher Erfüllungszeitpunkt des Events)
Der Erfüllungszeitpunkt eines Events nach einem gegebenen Zeitpunkt t_0 wird dann definiert durch

$$t_E^{t_0}(\sigma) := \min\{[t_0, \infty] \cap \mathfrak{M}_{E,\sigma}\}.$$

Wegen der vollständigen Ordnungsstruktur auf $\mathbb{R}$ ist dieser Wert eindeutig und wohl definiert.

Definition 3.6 (Startzeitpunkt der Phase)
Der Startzeitpunkt der Phase wird in dieser Situation definiert durch den frühest möglichen Erfüllungszeitpunkt, d.h.

$$t_\Phi^A(\sigma) := t_\Phi^{A,min}(\sigma) := \min\{[t_0, \infty] \cap \mathfrak{M}_{\Phi_{[d_1,d_2[},\sigma}\}.$$

Definition 3.7 (frühest und spätest möglicher Endzeitpunkt)
Der frühest mögliche Endzeitpunkt wird definiert durch

$$t_{\Phi}^{E,min}(\sigma) := t_{\Phi}^{A}(\sigma) + d_1$$

und der spätest mögliche Endzeitpunkt

$$t_{\Phi}^{E,max}(\sigma) := \min\{\neg\mathfrak{M}_{\Phi_{[d_1,d_2[},\sigma} \cap [t_{\Phi}^{A,min}(\sigma), \infty]\}$$

Man bemerke, dass $t_{\Phi}^{A,min} + d_1 < t_{\Phi}^{E,max}(\sigma) < t_{\Phi}^{A,min} + d_2$ *gelten muss.*

Auf diese Weise erhält man eine abstrahierte Zeitdarstellung des Signals σ, gegeben durch die frühesten Erfüllungszeitpunkte und ihren zugehörigen Eigenschaften. Ist ein Zeitpunkt dabei unendlich ist, erfüllt σ nicht das Verlaufsdiagramm.

Eventzeitpunkte und Eingangs- bzw. Ausgangszeitpunkte von Phasen werden im Folgenden auch kurz als Ereignisse bezeichnet.

Zusammengefasst: Falls die verwendeten Properties algorithmisch für ein Signal ausgewertet werden können und den obigen Einschränkungen genügen, lässt sich eine Verlaufsbeschreibung für Einzelsignale algorithmisch überprüfen.

Der skizzierte Algorithmus lässt sich auf temporale Implikationen und temporale Eigenschaften erweitern.

4 Signalabhängigkeiten

Die in Abschnitt 3 eingeführte Charakterisierung aufgrund einer Werteverlaufsanalyse erlaubt auch eine systematische Formulierung von Abhängigkeiten. Dazu soll hier etwas näher auf temporale Implikationen eingegangen werden. Mit ihrer Hilfe werden Zeitfenster für Merkmale durch ihre Abhängigkeit von weiteren Eigenschaften definiert.

Definition 4.1 (Temporale Implikationen)
Eine temporale Implikation ist gegeben durch die Angabe eines auslösenden Events oder Phase und eines davon abhängigen Ereignisses, sowie einem Mindest- und Höchstabstand $[d_1, d_2[\subset \mathbb{R}$*, der zeitlich zwischen auslösendem und abhängigen Ereignis liegen darf.*

Man kann natürlich hier auch halb-offene und offene Dauern spezifizieren. Es können Abhängigkeits-Eigenschaften in Switches boolesch verknüpft werden. Ferner sind auch negative Abstände erlaubt. In diesem Fall haben die abhängigen Ereignisse vorgängig stattzufinden.

Definition 4.2 (Prämissen / Konklusionszeitpunkt der Implikation)
Der Prämissenzeitpunkt einer Implikation ist definiert als der Zeitpunkt des in der Analyse des Einzelsignales gewonnenen zugehörigen Sourceelementes.

Der Konklusionszeitpunkt einer Implikation ist analog definiert als der Zeitpunkt des zugehörigen Targetelementes.

Definition 4.3 (Erfüllung der Implikation)
Sei t_I der bestimmte Prämissenzeitpunkt und $[d_1, d_2[\subset \mathbb{R}$ die spezifizierten Abstände, t_C der bestimmte Konklusionszeitpunkt. Dann wird die Implikation erfüllt g.d.w. gilt:

$$t_C \in [t_I + d_1, t_I + d_2[$$

Zur Erfüllung einer Implikation muss der (Start-) Zeitpunkt des Targetelements also innerhalb der Erfüllungsmenge der Implikation liegen.

Mit diesen vorgestellten Konstrukten und skizzierten Algorithmen zur Bewertung von Signalverläufen lassen sich auch bekannte Muster (*Pattern*) aus dem Bereich Model Checking formulieren und überprüfen, beispielsweise:

»Immer wenn Ereignis A eintritt, soll d_1 bis d_2 Zeitschritte später Ereignis B eintreten.«

Ereignis A kann dabei beliebig oft auftreten. Deshalb wird ein eigenes Diagramm für das Erkennen von A benötigt. Ebenso für Ereignis B. Die zeitliche Verknüpfung wird durch eine temporale Implikation $E_A^\sigma \overset{[d_1,d_2]}{\Rightarrow} E_B^\sigma$ formuliert (vgl. Abb. 6).

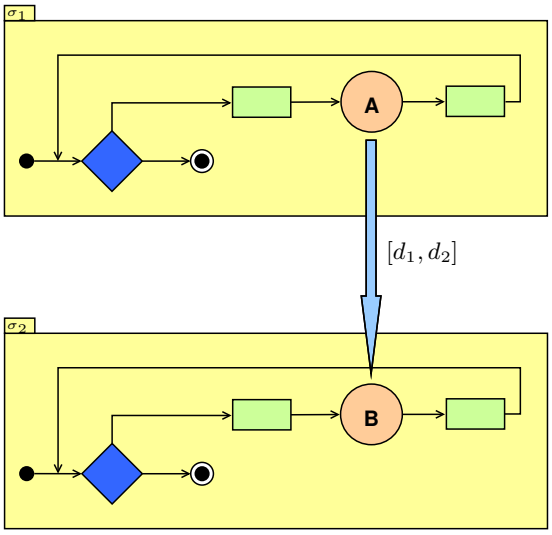


Abbildung 6: Pattern »Immer wenn A, dann nach $[d_1, d_2]$ Schritten auch B.«

Die zwischen den Events A und B geschalteten Phasen sind rein syntaktischer Natur (Phasen und Events müssen immer alternieren) und haben keinerlei weitere spezifische Eigenschaften. Mit diesen Konstrukten wird also entweder immer wieder A bzw. B erkannt oder man hat den Endzeitpunkt des Signals erreicht.

5 Zusammenfassung und Ausblick

Zur Unterstützung einer automatisierten Testauswertung in der modellbasierten Entwicklung eingebetteter Systeme wird in dieser Arbeit ein neuer Ansatz zur Beschreibung von kontinuierlichen Signalen vorgestellt. Anstelle eines konstruktiven, von ihrem zeitlichen Verlauf aufsetzenden Ansatzes wird hier von einer Analyse der Werteverläufe eines Signals ausgegangen und eine diskrete Beschreibung ihrer Verläufe vorgeschlagen. Die Modellierung des gewünschten Signalverlaufs erfolgt mit Hilfe von Zustandsautomaten. Diese sind dann algorithmisch zu überprüfen, wobei Zeitpunkte für die einzelnen Eigenschaften bestimmt werden.

Dieser Ansatz erlaubt in systematischer Weise auch die Beschreibung von Signalabhängigkeiten. Dazu werden die zuvor bestimmten Eigenschafts- und Diagramm-Zeitpunkte der Signale als Bezugspunkte verwendet. Auf diese Weise erhält man einen sehr allgemeinen Ansatz zur Beschreibung von Signalverläufen und ihren Abhängigkeiten, der zugleich eine algorithmische Auswertung erlaubt.

In einem folgenden Artikel sollen die hier nur skizzierten Ansätze ausgeführt und weiterentwickelt werden ([GW07]). Ferner ist eine prototypische Implementierung dieser Algorithmen geplant.

Literatur

[AHLP00] R. Alur, T.A. Henzinger, G. Lafferriere und G.J. Pappas. Discrete Abstractions of Hybrid Systems. In *Proceedings of the IEEE*, 88, Seiten 971–984, 2000.

[Con04] M. Conrad. *Auswahl und Beschreibung von Testszenarien für den Modell-basierten Test eingebetteter Software im Automobil*. Dissertation, TU Berlin, 2004.

[GG93] M. Grochtmann und K. Grimm. Classification Trees for Partition Testing. *Software Testing, Verification & Reliability (STVR)*, 3(2):63–82, 1993.

[GHK+06] B.S. Gulavani, T.A. Henzinger, Y. Kannan, A.V. Nori und S.K. Rajamani. SYNERGY: A New Algorithm for Property Checking. In *Proceedings of SIFSOFT 06/FSE-14*. ACM, 2006.

[GW07] C. Gips und H.-W. Wiesbrock. Konzeption eines Tools zur automatischen Testauswertung von Hard- und Softwaretests. in preparation, 2007.

[Leh03] E. Lehmann. *Time Partition Testing – Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen*. Dissertation, TU Berlin, 2003.

[mev] MEval. `www.itpower.de/meval.html`.

[ZNSP06] J. Zander-Nowicka, I. Schieferdecker und A.M. Pérez. Automotive Validation Functions for On-line Test Evaluation of Hybrid Real-time Systems. In *Proceedings of IEEE 41st Anniversary of the Systems Readiness Technology Conference (AutoTestCon 2006) in USA*, 2006.

Automatisierte, werkzeugübergreifende Richtlinienprüfung zur Unterstützung des Automotive-Entwicklungsprozesses

Tibor Farkas[1], Harald Röbig[2]

[1] Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin

tibor.farkas@fokus.fraunhofer.de

[2] CARMEQ GmbH
Carnotstr. 4
10587 Berlin

harald.roebig@carmeq.com

Abstract: Die Entwicklung von eingebetteten Systemen im Automobilbereich ist gekennzeichnet von steigender Komplexität der zu entwickelnden Fahrzeugfunktionen. Die Handhabung dieser Komplexität wird über entsprechend mächtige Werkzeuge, welche die verschiedenen Aktivitäten des Systementwicklungsprozesses unterstützen, angestrebt. Als eine Schwachstelle erweist sich dabei die fehlende Durchgängigkeit der eingesetzten Werkzeugketten im Automobilbereich, die somit eine werkzeugübergreifende Konsistenz der jeweils erzeugten Artefakte nicht garantieren können. Infolgedessen sind in der Entwicklungspraxis häufig Schnittstelleninkonsistenzen oder fehlerhafte bzw. nicht nachvollziehbar umgesetzte Anforderungen zu beobachten. Das Forschungsprojekt MESA[1] adressiert dieses Problem, indem es ein metamodellbasiertes Werkzeug zur automatischen Konsistenzsicherung von Entwicklungsartefakten über Werkzeuggrenzen hinweg erarbeitet, das leicht an verschiedene Entwicklungsprozesse und –werkzeuge anpassbar ist. Das vorliegende Papier beschreibt die in MESA angestrebte werkzeugübergreifende Konsistenzprüfung und stellt die bereits erzielten Ergebnisse an einem Anwendungsbeispiel vor.

1 Einleitung

In modernen Fahrzeugen wird eine rasant wachsende Zahl von Funktionen durch elektronische Bauteile realisiert, um wettbewerbsdifferenzierende Innovationen zu schaffen. Der Anteil von Fahrzeugfunktionen, die dabei durch Software gesteuert werden, nimmt hierbei kontinuierlich zu. Die Beherrschung der durch die zunehmend vernetzten Funktionen bedingten Systemkomplexität wird für Fahrzeughersteller und deren Zulieferer zu einem zentralen, wettbewerbsentscheidenden Faktor. Von größter

[1] MESA — Metamodellierung zur Automatisierung von Analyse- und Entwicklungsmethoden für Software im Automobil. Dieses Vorhaben wird von der europäischen Union und vom Land Berlin kofinanziert (Europäischer Fonds für Regionale Entwicklung).

Wichtigkeit sind dabei sorgfältig ausgewählte Entwicklungsmethoden und -werkzeuge, da nur effiziente und Fehler vermeidende Prozesse diesen Wettbewerbsvorteil sicherstellen. Problematisch bezüglich der Fehlervermeidung erweist sich, dass die zur Verfügung stehenden Werkzeugketten im Automobilbereich keine *übergreifende* Konsistenz der in den jeweiligen Werkzeugen erzeugten Artefakte garantieren können. Konsistenzprüfungen werden gegebenenfalls von Entwicklern lokalisiert und semi-automatisiert in den einzelnen Werkzeugen durch Batch-Skripte in der jeweils proprietären Programmiersprache des Werkzeugherstellers durchgeführt. Aufgrund dieses Sachverhalts sind in der Praxis häufig Probleme durch Schnittstelleninkonsistenz, fehlerhaft bzw. nicht nachvollziehbar umgesetzte Anforderungen, keine Standardisierung der Richtlinien und fehlende Durchgängigkeit zu beobachten.

1.1 Übersicht über die Kapitel

In *Kapitel 1* führen wir den in MESA betrachteten modellbasierten Entwicklungsprozess ein und zeigen, warum eine automatisierte, werkzeugübergreifende Konsistenzsicherung den bestehenden Entwicklungsprozess entscheidend verbessert.

In *Kapitel 2* stellen wir unser Vorgehen beim Entwickeln eines metamodellbasierten Tools für Konsistenzchecks vor und zeigen, dass dieser Ansatz die Übertragbarkeit auf beliebige Entwicklungsprozesse sehr einfach gestaltet.

In *Kapitel 3* führen wir ein einfaches Beispiel für eine Konsistenzprüfung vor und demonstrieren die prototypische Umsetzung des so genannten ASD[2] Regelcheckers.

1.2 Konsistenzsicherung über verschiedene Entwicklungsphasen

Bisher sind Systementwickler, die sich um Einhaltung der übergreifenden Konsistenz von Entwicklungsartefakten (z.B. Dateien, Dokumenten) bemühen, auf sich alleine gestellt. Das Erfüllen von Konsistenzkriterien kann von ihnen oftmals nur durch manuelles Vergleichen einer großen Menge von Daten erreicht werden. Diese Tätigkeiten geschehen manuell, da sie typischerweise einen Übergang zwischen Artefakten betreffen, die mit verschiedenen Werkzeugen erstellt werden. Eine nähere Betrachtung der Tätigkeiten, die zur werkzeugübergreifenden Konsistenzsicherung durchgeführt werden müssen, zeigt, dass diese bezüglich des intellektuellen Anspruchs meistens sehr einfach sind. Die Belastung für den Entwickler entsteht durch die große Anzahl an Wiederholungen, mit der er diese Tätigkeiten ausführen muss. Gerade solcherart Tätigkeiten sind es aber auch, die sich besonders gut automatisieren lassen. Wir versprechen uns daher vom Einsatz eines automatisierten, werkzeugübergreifenden Konsistenzcheckers einen hohen Produktivitäts- und Qualitätsgewinn der Entwicklungstätigkeit.

---

[2] ASD – Automotive System Development

Das Projekt MESA betrachtet die grundlegenden Phasen des V-Modells, d.h. die Anforderungsanalyse, den Systementwurf, die Implementierung und den Systemtest. Die Auswahl der zu unterstützenden Werkzeuge bezieht sich in diesem Projekt auf den modellbasierten Entwicklungsprozess mit dem zentralen Werkzeug MATLAB/Simulink/Stateflow (ML/SL/SF) [MAT06]. Die angewendete Methodik unterstützt prinzipiell jede mögliche Werkzeugkette, sofern die einzelnen Werkzeuge grundlegende Informationsschnittstellen oder offene Dateiformate bereitstellen.

1.3 Der modellbasierte Entwicklungsprozess

Der zu Beginn des modellbasierten Entwicklungsprozesses stehende Schritt ist die Entwicklung von Anforderungen für Fahrzeugfunktionen. Diese werden von den Anforderungen aus Stakeholder-Sicht ausgehend, auf funktionaler, logischer Ebene in so genannten Basisfunktionen strukturiert erfasst. Dabei kommt das Werkzeug DOORS [TLG06] zum Einsatz. Auf logischer Ebene abstrahiert die Funktionsbeschreibung vollständig von der späteren Umsetzung in einem Steuergerätenetzwerk und angeschlossener Sensorik und Aktorik. Basisfunktionen haben Eingangs- und Ausgangsschnittstellen und beschreiben, wie mit Hilfe von Parametern die logischen Eingaben in logische Ausgaben verwandelt werden (Abbildung 1.1 zeigt ein Beispiel). Die als Eingangs- und Ausgangsgrößen verwendeten Signale werden in einer logischen Datendefinition näher beschrieben. Sie verknüpfen die Basisfunktionen zu Funktionsnetzwerken.

5 **3.1.1 Basisfunktion SCHEIBENWISCHER_AKTIVIEREN**
6 **3.1.1.1 Beschreibung**
8 Diese Funktion aktiviert den Scheibenwischer, nach Zündung ein.
7 **3.1.1.2 Eingaben**
9 s_zuendung_ein
s_benutzerw_scheibenw_aktivieren
10 **3.1.1.3 Ausgaben**
11 s_scheibenw_aktivieren
12 **3.1.1.4 Parameter**
13 **3.1.1.5 Verarbeitung**
15 **Bedingung**:
- s_zuendung_ein UND
- s_benutzerw_scheibenw_aktivieren

Aktion:
- s_scheibenw_aktivieren

14 **3.1.1.6 Fehlerbehandlung und Diagnose**

Abbildung 1.1 - Ausschnitt aus einer in DOORS beschriebenen Basisfunktion

Mit Hilfe der Verhaltensmodellierung in ML/SL/SF kann die korrekte Spezifikation des Basisfunktionsnetzwerks und der logischen Signale und Parameter frühzeitig validiert werden, denn die erzeugten Modelle sind simulierbar. Die korrekte Umsetzung der Anforderungen in die Verhaltensmodelle wird mit Hilfe von anforderungsbasierten Testfällen verifiziert, die mit Hilfe des Werkzeugs MTest inklusive CTE/ES [WEW05]

ausgeführt und ausgewertet werden. Der nächste Schritt des modellbasierten Entwicklungsprozesses ist die Weiterentwicklung der Verhaltensmodelle zu Implementierungsmodellen, die – optimiert für die jeweilige Zielplattform – zu automatisch generiertem Code führen.

In der nachfolgenden Abbildung 1.2 ist ein typischer modellbasierter Entwicklungsprozess skizziert. Jeder Phase sind die bereits genannten Werkzeuge exemplarisch zugeordnet. In dieser wie auch anderen Werkzeugketten liegen, aufgrund diverser Werkzeughersteller, Informationen in meist inkompatiblen werkzeugspezifischen Dateiformaten vor, die eine übergreifende Konsistenzsicherung der mit den Werkzeugen bearbeiteten Entwicklungsdaten erschweren.

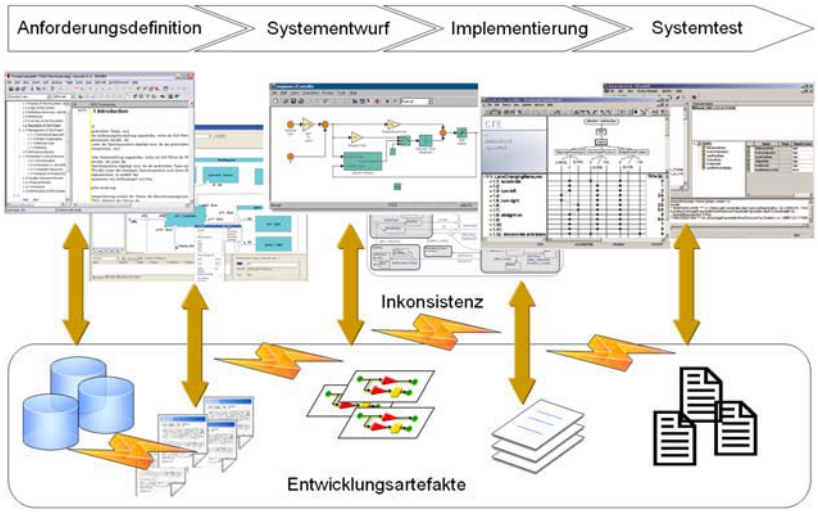


Abbildung 1.2 - Werkzeugkette der modellbasierten Automotive-Systementwicklung
Fehlende Durchgängigkeit führt zu potenziell inkonsistenten Entwicklungsartefakten

Für einen durchgängigen und konsistenten Entwicklungsprozess stellt sich die Frage, in wie weit sich isolierte Konsistenzprüfungen von einzelnen Werkzeugen und Entwicklungsartefakten übergreifend und für eine Automatisierung hinreichend formal beschreiben lassen. Ziel des Projektes MESA ist es daher, ein Verfahren für eine werkzeugübergreifende und automatisierte Konsistenzsicherung von Entwicklungsartefakten zur Unterstützung aktueller Entwicklungsprozesse zur Verfügung zu stellen [MSA06].

2 Vorgehensweise zur Automatisierung der Konsistenzsicherung

Für eine durchgängige und phasenübergreifende Konsistenzsicherung aller in verschiedenen Entwicklungsphasen erzeugten Artefakte müssen die jeweils gelebten Entwicklungsprozesse analysiert, verstanden und abgebildet werden. Voraussetzung

einer automatisierten Konsistenzsicherung ist eine Werkzeugunterstützung jeder einzelnen Entwicklungsphase. In diesem Kapitel wird unsere Vorgehensweise zur Automatisierung der Konsistenzsicherung inklusive der verwendeten Werkzeugkette dargestellt.

2.1 Gemeinsames Metamodell für alle Entwicklungswerkzeuge

Die im Forschungsprojekt MESA gewählte Model Driven Architecture [OMG2] der Object Management Group (OMG) [OMG1] definiert und standardisiert zur Problemlösung einen modellbasierten Ansatz.

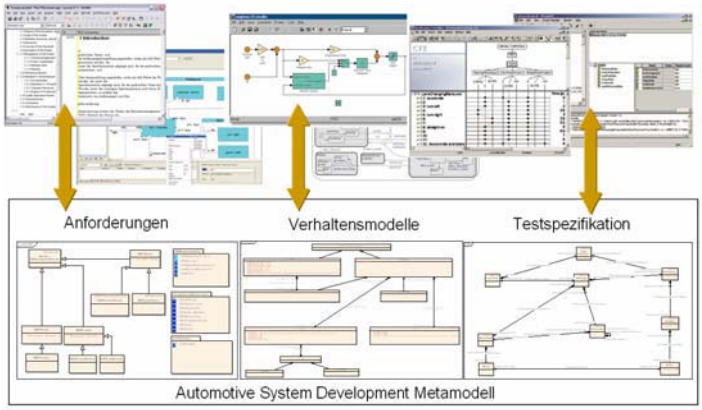


Abbildung 2.1 - Das ASD-Metamodell stellt die innere Struktur der Artefakte in den Entwicklungswerkzeugen dar

Abbildung 2.1 zeigt einen der ersten notwendigen Schritte unserer Vorgehensweise. Die innere Struktur der Artefakte in den unterschiedlichen Werkzeugen wird in einem gemeinsamen Metamodell abgebildet. Dazu nutzen wir das CASE-Werkzeug Enterprise Architect [SPX06] für die Erstellung von Metamodellen nach dem MOF Standard [OMG3]. Nachfolgend wird der Aufbau des Metamodells kurz skizziert.

Das ASD-Metamodell ist hierarchisch nach Abstraktionsebenen aufgebaut. Es definiert zunächst einen Kern, der ein Standardelement (ASDElement) sowie wiederkehrende Strukturen, wie z.B. Containment, beschreibt. Die Kernelemente werden in den weiteren Paketen des Metamodells durch Vererbung wieder verwendet.

Das ASD-Metamodell enthält neben dem Paket mit grundlegenden Elementen zwei weitere Arten von Paketen: Strukturen von Werkzeugartefakten und Strukturen logischer Artefakte. Logische Artefakte entstehen typischerweise durch eine Abstraktion vom Werkzeug. Ein typisches Beispiel für ein logisches Artefakt ist die bereits beschriebene Basisfunktion (siehe Abbildung 2.2). Eine Basisfunktion wird im Werkzeug DOORS durch eine bestimmte hierarchische Strukturierung von Anforderungs-Objekten dargestellt (siehe Abbildung 1.1).

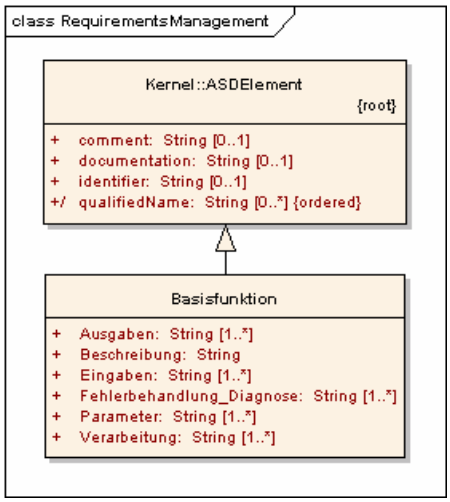


Abbildung 2.2 - Das logische Artefakt "Basisfunktion"

2.2 Gemeinsames Repository und Werkzeugadapter

Abbildung 2.3 zeigt das Paket des ASD-Metamodells, das die modellierten Werkzeugartefakte enthält. Direkt aus Enterprise Architect heraus lässt sich mit Hilfe des Werkzeugs medini meta modeler [HAN06; IKV06] aus dem gesamten ASD-Metamodell ein MOF-konformes Repository generieren, welches als Server-Dienst über das Netzwerk ansprechbar ist und Instanzen der modellierten Werkzeugartefakte aufnehmen kann.

Um die Informationen aus den Werkzeugen in das Repository zu übertragen, wurden von uns integrierte Werkzeugadapter entwickelt. Bisher werden Anforderungsprojekte aus DOORS und ML/SL/SF-Modelle [DAB04] aus dem MATLAB-Workspace [HAL05], jeweils über die COM-Schnittstelle [MCT06] der Werkzeuge ausgelesen. Die Einbindung von Testspezifikationen in CTE/ES erfolgt durch XML [W3C6], da dieses Werkzeug den COM-Standard nicht unterstützt. Die ausgelesenen Informationen werden mittels CORBA [OMG5] über das Netzwerk als eine Modellinstanz im Repository gespeichert (Client-Server-Architektur). Das gespeicherte Instanzmodell entspricht exakt der Struktur des vorgegebenen Metamodells.

Logische Artefakte, wie z.B. das Konstrukt *Basisfunktion*, sind in den Werkzeugen nicht als solches vorhanden. Sie können daher auch nicht direkt aus den Werkzeugen ausgelesen werden, sondern entstehen nachträglich durch Transformationen auf den Instanzmodellen eines oder mehrerer Werkzeugartefakte im Repository. Diese Transformationen werden von uns zurzeit spezifiziert und implementiert.

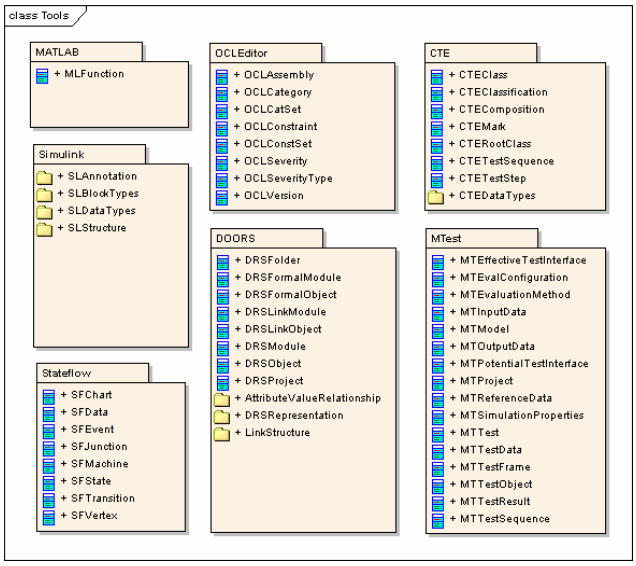


Abbildung 2.3 - Ausschnitt (Paket Tools) aus dem Metamodell *Automotive System Development* (ASD).

2.3 Formalisierung und Prüfung werkzeugspezifischer sowie werkzeugübergreifender Entwicklungsrichtlinien

Auf dem in MESA entstandenen ASD-Metamodell können natürlichsprachliche Entwicklungsrichtlinien mit Hilfe der OCL [OMG4] formalisiert werden. Eine Softwarekomponente [OSL06] prüft dann automatisiert und werkzeugunabhängig die formalisierten Ausdrücke auf einem oder mehreren Instanzmodellen in dem Modell-Repository. Der vorgestellte Ansatz erlaubt es, sowohl Richtlinien für Artefakte einzelner Werkzeuge als auch werkzeugübergreifende Richtlinien zu prüfen. Da OCL-Ausdrücke Modellinstanzen nicht verändern können, lassen sich einerseits beliebig viele Ausdrücke auf einem Modell parallel prüfen und andererseits mit einem Ausdruck beliebig viele Modellinstanzen parallel durchlaufen, was diese Lösung gut skalierbar macht.

Wie eingangs beschrieben, ist von uns exemplarisch der modellbasierte Entwicklungsprozess zur Eingrenzung der Problemdomäne ausgewählt worden. In diesem Prozess sind bereits Richtlinien üblich, die Artefakte in einzelnen Entwicklungsphasen betreffen. Dies sind Modellierungsrichtlinien für ML/SL/SF und formale Reviewkriterien für DOORS-Lastenhefte und Testspezifikationen. Eine einfache werkzeugübergreifende Richtlinie wird in Kapitel 3.1 beschrieben. Die Formulierung werkzeugübergreifender Richtlinien ist nur nach gründlicher Analyse des Entwicklungsprozesses möglich. Hier sehen wir für die Kunden eines möglichen zukünftigen Produktes ASD Regelchecker Bedarf an Beratung. Typischerweise hat die Formulierung neuer Richtlinien auch Auswirkungen auf das Metamodell, da

prozessspezifische logische Artefakte zu definieren sind. Werkzeugadapter sind lediglich beim Einbinden bisher nicht berücksichtigter Werkzeuge zu programmieren. Grundsätzlich sind Werkzeugadapter prozessunspezifisch und damit wiederverwendbar implementiert.

Eine auf dem .NET Framework 2.0 [MNT06] basierte Applikation (ASD Regelchecker) [MSA06] fasst die zuvor geschilderten Komponenten unter einer gemeinsamen grafischen Benutzeroberfläche zusammen. In Abbildung 2.4 ist ein Ausschnitt der technischen Architektur des ASD Regelcheckers am Beispiel des ML/SL/SF-Werkzeugadapters dargestellt. Die technische Anbindung weiterer Entwicklungswerkzeuge, wie der bereits angebundenen Werkzeuge DOORS, ML/SL/SF und CTE/ES, funktioniert weitestgehend analog.

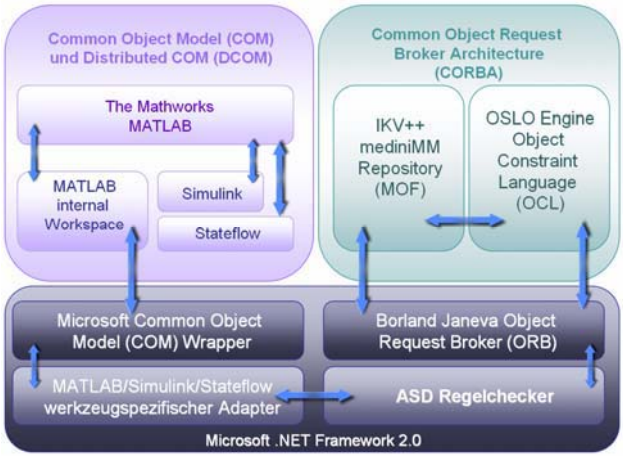


Abbildung 2.4 - Architektur und Technologien des ASD Regelcheckers am Beispiel des Werkzeugadapters für MATLAB/Simulink/Stateflow

3 Metamodellbasierte Regelbeschreibung und Überprüfung

Wie bereits in Abschnitt 2 dargestellt, ist eine genaue Kenntnis des zu unterstützenden Entwicklungsprozesses unabdingbar. Dieses Wissen kommt im Wesentlichen an zwei zentralen Punkten der Entwicklung eines Konsistenzcheckers zum Einsatz: Zum einen müssen die zu prüfenden Artefakte, die meist logische Artefakte sind, im Metamodell definiert werden und ihre Zusammensetzung aus Werkzeugartefakten mit Hilfe von Transformationen dargestellt werden; zum anderen fließt das Prozesswissen in die Neuformulierung oder zumindest Anpassung bestehender Entwicklungsrichtlinien ein. Im Folgenden stellen wir anhand des von uns untersuchten Entwicklungsprozesses beispielhafte Entwicklungsrichtlinien dar.

3.1 Beispiel einer werkzeugübergreifenden Konsistenzprüfung

Im diesem Abschnitt wird anhand der Umsetzung einer Basisfunktion aus DOORS in ein Simulink Verhaltensmodell ein übersichtliches Beispielszenario des in MESA entwickelten ASD Regelcheckers erläutert.

Abbildung 1.1 zeigt exemplarisch einen Ausschnitt einer funktionalen Anforderung an die Steuerung eines Scheibenwischers. Abbildung 3.1 zeigt das entsprechende Subsystem in MATLAB/Simulink, das die dargestellte Anforderung in einem Verhaltensmodell realisiert. Da das Simulink-Subsystem die Funktionalität einer Basisfunktion kapselt, wird es *Basismodul* genannt.

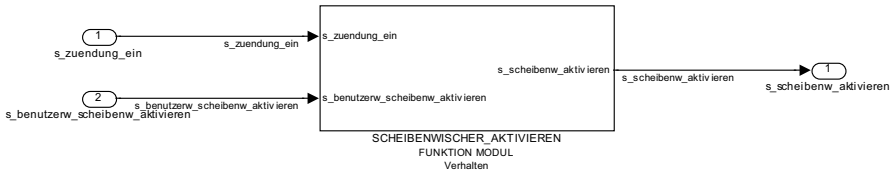


Abbildung 3.1 - Ausschnitt aus einem Verhaltensmodell: Ein Basismodul (Simulink-Subsystem), das die Funktionalität zur Umsetzung einer in DOORS beschriebenen Basisfunktion kapselt

Der Modellierer hat nun sicherzustellen, dass er zu jeder Zeit (d.h. bei jeder neuen Version des Lastenheftes in DOORS) alle Basisfunktionen in Simulink Basismodulen modelliert hat. Um diese Prüfung zu erleichtern, gilt die Richtlinie, dass die Bezeichnung der Basisfunktion in DOORS mit der Bezeichnung des Basismoduls in Simulink übereinstimmen soll.

Weitere automatisch prüfbare Entwicklungsrichtlinien sind beispielsweise, dass die Basismodule die gleichen Schnittstellen haben, wie in den Basisfunktionsanforderungen vorgegeben wird, und dass die Schnittstellen die gleichen Datentypen erwarten, wie es in der logischen Datendefinition definiert wurde. Es sei an dieser Stelle ausdrücklich darauf hingewiesen, dass die Überprüfung, ob die Simulink/Stateflow-Modelle die in den Anforderungen beschriebenen Funktionalitäten abbilden, auch weiterhin dem Modellierer und dem anschließenden Test überlassen bleiben.

Die Anforderung der vollständigen Umsetzung aller Basisfunktionen in Basismodule lässt sich, basierend auf dem beschriebenen ASD-Metamodell, folgendermaßen in OCL formalisieren:

```
context OclVoid inv:AutomotiveSystemDevelopment::Activities::
RequirementsManagement::Basisfunktion.allInstances()->reject
(b|AutomotiveSystemDevelopment::Tools::Simulink::SLBlockTypes::
SLSubSystemBlock.allInstances()->exists(s|s.identifier=b.identifier))
```

Dieser OCL-Ausdruck besagt Folgendes: „*Aus allen Instanzen b der Klasse ‚Basisfunktion' behalte nur diese, für die* ***nicht*** *gilt, dass es eine Instanz s der Klasse ‚SLSubSystemBlock' mit dem gleichen Bezeichner gibt.*".

Das Ergebnis der Auswertung dieses Ausdrucks ist somit eine Menge von Basisfunktionen, für die es noch keine entsprechenden Subsysteme in Simulink gibt. Ist die Menge leer, so existieren für alle Basisfunktionen Basismodule mit dem gleichen Bezeichner.

Nach anfänglicher Formulierung der Richtlinien mit Wahrheitswerten als Rückgabetypen, favorisieren wir inzwischen eine mengenorientierte Formulierung. Das Ergebnis der Auswertung solcher Richtlinien lässt sich dann wie folgt interpretieren: Ist das Ergebnis eine leere Menge, so ist die Regel eingehalten; ist das Ergebnis eine nicht leere Menge, so existieren Verstöße gegen die Regel. Die Objekte, die gegen die Regel verstoßen, sind genau die in der Rückgabemenge enthaltenen Objekte.

3.2 Automatische Konsistenzprüfung in der Anwendung

Im folgenden Abschnitt soll kurz auf die bereits erfolgte prototypische Umsetzung des von uns ASD Regelchecker genannten Tools eingegangen werden.

Mittels im Werkzeug integrierter Menüs kann der ASD Regelchecker aus der jeweiligen Applikation (DOORS oder Simulink) kontextbasiert aufgerufen werden. So können beispielsweise bei Bedarf einzelne Artefakte des gerade verwendeten Werkzeugs selektiv geprüft werden. Der Regelchecker kann zudem auch als eine separate Einzelapplikation gestartet werden. Vor dem Prüfdurchlauf lädt der Regelchecker die in OCL formulierten Richtliniendefinitionen und erlaubt die Selektion von kategorisierten Richtlinienprofilen.

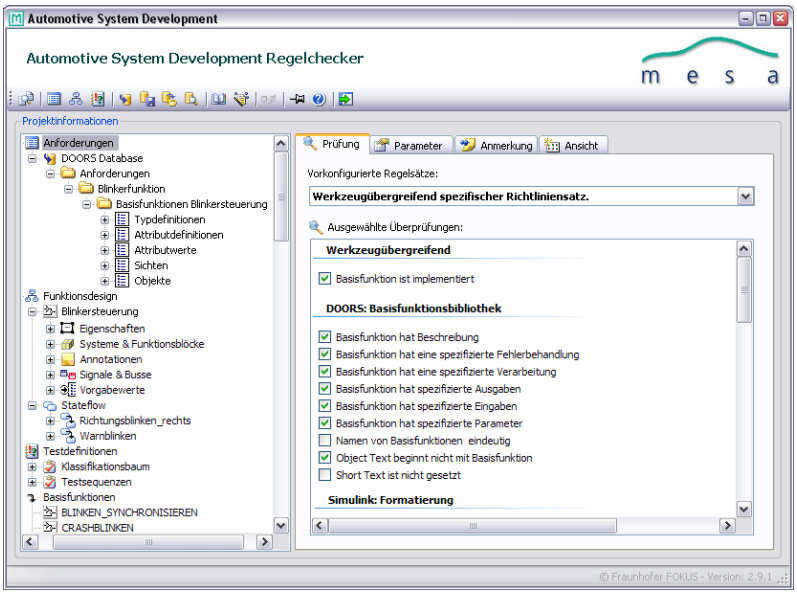


Abbildung 3.2 - Bedienoberfläche des ASD Regelcheckers zur Konfiguration der Regelprüfung

Abbildung 3.2 zeigt den Aufbau der grafischen Benutzeroberfläche. Im linken Teil der Oberfläche wird ein hierarchischer Artefaktbaum mit den gewählten Anforderungen aus DOORS und dem gewählten Modell aus ML/SL/SF aufgebaut. Elemente können im Nachhinein vom Benutzer angepasst oder aber entfernt werden, sofern sie von der Prüfung ausgeschlossen werden sollen. Im rechten Teil der Oberfläche befinden sich mehrere Registerkarten für Einstellungsoptionen. In der Registerkarte *Prüfung* sind die prüfbaren Richtlinien aufgelistet. Nach erfolgter Konfiguration wird die Konsistenzprüfung durch den Benutzer gestartet.

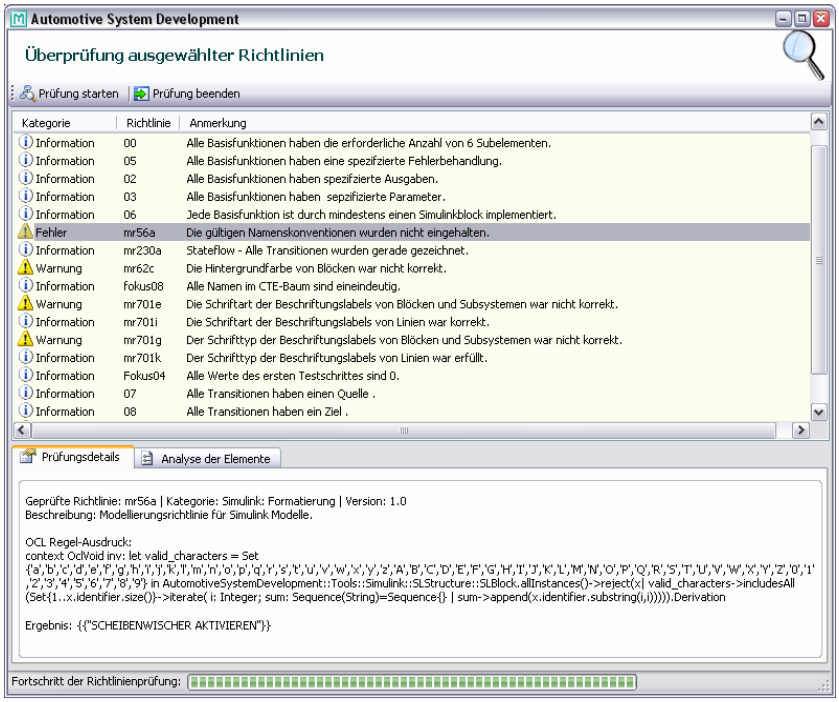


Abbildung 3.3 – Darstellung der Ergebnisse einer abgeschlossenen Richtlinienprüfung

Abbildung 3.3 zeigt einen Ausschnitt des ASD Regelcheckers für die Analyse von Prüfergebnissen nach einem erfolgten Prüfdurchlauf. Ein Ergebnisprotokoll in Listenform gestattet es dem Entwickler herauszufinden, welche Richtlinien erfüllt und welche gegebenenfalls verletzt wurden. Die im Ergebnisprotokoll selektierten Informationen, Hinweise oder Fehler in einem speziellen Ausgabefeld („Prüfungsdetails“) ausführlich beschrieben. Hier wird auch gezeigt welche konkreten Elemente für die Regelverletzung verantwortlich sind (im Beispiel ist das der Simulink-Block *SCHEIBENWISCHER AKTIVIEREN*). Die Registerkarte „Analyse der Elemente“ ermöglicht es, dass diese Elemente im Ursprungswerkzeug markiert und direkt angesprungen werden können. (Bisher nur für Simulink implementiert).

4 Zusammenfassung und Ausblick

Dieser Beitrag stellt das Forschungsvorhaben des Projektes MESA zur werkzeugübergreifenden Konsistenzsicherung von Entwicklungsartefakten dar. Die betrachteten Werkzeuge sind DOORS, MATLAB/Simulink/Stateflow und MTest mit CTE/ES. Der Lösungsansatz beruht auf der Abbildung werkzeugspezifischer Artefakte in ein MOF konformes Metamodell. Die Konsistenzprüfung wird auf den Instanzen des Metamodells anhand der OCL vorgenommen. Dieser skalierbare Ansatz erlaubt gleichermaßen die Überprüfung werkzeugübergreifender wie werkzeugspezifischer Richtlinien, z.B. der Modellierungsrichtlinien für MATLAB/Simulink/Stateflow. Die Praxistauglichkeit des Ansatzes wird durch ein prototypisch entwickeltes Softwarewerkzeug „ASD Regelchecker“ demonstriert.

Neben der technologischen Anbindung weiterer Werkzeuge und der Implementierung von Transformationen liegt der inhaltliche Schwerpunkt weiterer Arbeiten auf der Formalisierung zunehmend komplexerer Entwicklungsrichtlinien.

5 Literaturverzeichnis

[DAB04] Dabney, J.; Harman, T.: *Mastering Simulink*, Pearson Education, Prentice Hall, 2004

[HAL05] Hanselman, D.; Littlefield, B.: *Mastering Matlab 7*, Pearson Education, 2005

[HAN06] Publikation in hanser automotive electronics+systems: *Schnellere Softwareentwicklung-Optimierter Entwicklungsprozess*, Ausgabe: 1-2/2006, Carl Hanser Verlag, München

[IKV06] IKV++ Technologies AG: *Medini Meta Modeler*, URL: www.ikv.biz

[MAT06] The MathWorks Inc., *MATLAB/Simulink/Stateflow*, URL: www.mathworks.com

[MSA06] Forschungsprojekt des FhI FOKUS und der Carmeq GmbH: *Metamodellierung zur Automatisierung von Analyse- und Entwicklungsmethoden für Software im Automobil,* URL: www.fokus.fraunhofer.de/bereichsseiten/projekte/MESA

[MCT06] Microsoft: *Component Object Model Technologies*, URL: www.microsoft.com/com

[MNT06] Microsoft: *.NET Framework*, URL: www.microsoft.com/germany/msdn/netframework

[OMG1] Object Management Group (OMG), URL: www.omg.org

[OMG2] Object Management Group (OMG*): Model Driven Architecture (MDA)*, URL: www.omg.org/mda

[OMG3] Object Management Group (OMG): Meta Object Facility (MOF), Version 1.4, URL: www.omg.org/technology/documents/formal/mof.htm

[OMG4] Object Management Group (OMG): *Object Constraint Language 2.0 Specification*, URL: www.omg.org/docs/ptc/05-06-06.pdf

[OMG5] Object Management Group: *Common Object Request Broker Architecture (CORBA)*, OMG-Document formal/2004-03-12

[OSL06] OSLO – Open Source Library for OCL, URL: oslo-project.berlios.de

[SPX06] Sparx Systems Ltd.: *Enterprise Architect*, Version 6.1, URL: www.sparxsystems.com

[TLG06] Telelogic Deutschland GmbH: *DOORS*, Version 8.0, URL: www.telelogic.com

[WEW05] Wewetzer, C.: *MTest – eine offene Testumgebung für die modellbasierte Entwicklung, Abteilung Produktmanagement*, Design und Elektronik Entwicklerforum, 2005

[W3C6] Extensible Markup Language, World Wide Web Consortium, URL www.w3.org

Typisierung und Verifikation zeitlicher Anforderungen automotiver Software Systeme

Matthias Gehrke, Martin Hirsch, Wilhelm Schäfer
Software Quality Lab (s-lab)
Universität Paderborn
D-33095 Paderborn
[mgehrke|mahirsch|wilhelm]@upb.de

Oliver Niggemann, Dirk Stichling
dSPACE GmbH
Technologiepark 25
D-33100 Paderborn
[ONiggemann| DStichling]@dspace.de

Ulrich Nickel
Hella KGaA
Rixbecker Str. 75
D-59552 Lippstadt
Ulrich.Nickel@hella.com

Zusammenfassung: Ein wesentliches Problem der Integration von Steuergeräten im Automobil besteht darin, dass die einzelnen Steuergeräte in ihrer Funktionalität zwar spezifiziert werden, dass aber mögliche Inkompatibilität und daraus resultierende Fehler insbesondere im Hinblick auf zeitliche Anforderungen an das Gesamtsystem erst während des Integrationstest erkannt werden (können). Wir schlagen ein modellbasiertes Vorgehen vor, das basierend auf einer AUTOSAR-kompatiblen Komponentenarchitektur die Spezifikation der Funktionalität und der zeitlichen Randbedingungen unterstützt und darüber hinaus die komponentenübergreifende automatische Überprüfung von zeitlichen Anforderungen an das Gesamtsystem auf der Basis der Modelle der einzelnen Steuergeräte ermöglicht.

1 Einleitung

Die in Kraftfahrzeugen eingesetzten Steuergeräte werden insbesondere durch den verstärkten Einsatz von Software immer leistungsfähiger. Darüber hinaus wird durch einen massiven Ausbau ihrer Vernetzung untereinander wesentlich weitergehende Funktionalität und damit zusätzlicher Komfort und mehr Sicherheit möglich und realisiert. Ein Beispiel für diesen Trend sind moderne Fahrerassistenzsysteme. Für einen Automobil-Zulieferer wie die Hella KGaA Hueck & Co führt dieser bekannte Trend zu immer größeren Projekten und vor allem immer umfangreicher werdenden Komponentenspezifikationen, da immer komplexere Funktionen in die einzelnen Steuergeräte zu integrieren sind. Bei Herstellern wird der Aufwand für die Integration der einzelnen Steuergeräte dementsprechend immer aufwendiger und auch schwieriger.

Ein wesentliches Problem dieser Integration besteht darin, dass die einzelnen Steuergeräte in ihrer Funktionalität zwar spezifiziert werden, dass aber oft eine genaue Spezifikation der zeitlichen Abläufe und Restriktionen fehlt oder nur in Form von natürlich sprachlichem Text ausgedrückt wird. Selbst wenn diese Spezifikation in formaler Notation vorliegt, sind komponentenübergreifenden Prüfungen auf Widerspruchsfreiheit oder Vollständigkeit aufgrund der Komplexität solcher Systeme oftmals nur schwer möglich. Bei der Integration der Steuergeräte, d.h. beim ersten Test stellt sich dann erst heraus, ob die in den einzelnen Steuergeräten realisierten Funktionen und ihre zeitlichen Randbedingungen miteinander kompatibel sind und nicht sogar zu Fehlfunktionen führen (können).

Wir schlagen ein modellbasiertes Vorgehen vor, dass basierend auf einer AUTOSAR-kompatiblen Komponentenarchitektur die Spezifikation der Funktionalität und der zeitlichen Randbedingungen unterstützt und darüber hinaus die komponentenübergreifende automatische Überprüfung von zeitlichen Anforderungen an das Gesamtsystem auf der Basis der Modelle der einzelnen Steuergeräte und somit weit vor einer Testphase ermöglicht.

Im folgenden Kapitel wird hierzu das Konzept des so genannten „Master"-Automaten vorgestellt. Dieser stellt die für eine Überprüfung einer Eigenschaft des Gesamtsystems minimal notwendige Verschaltung der einzelnen Automaten, die die Funktionalität eines einzelnen Steuergerätes bzw. einer einzelnen Komponente spezifizieren, dar. In Kapitel 3 wird gezeigt, wie dieser Automat auf der Basis einer einfachen parametergesteuerten Eingabe der vom Gesamtsystem gewünschten Eigenschaft automatisch erzeugt werden kann und wie diese gewünschte komponentenübergreifende Eigenschaft dann automatisch durch Model Checking überprüft wird. Zusammenfassung und Ausblick in Kapitel vier schließen das Papier ab.

2 Komponentenbasierter Entwurf

Die Architekturmodellierung eines automotiven Softwaresystems in [GH+06] basiert auf einem Komponentenansatz gemäß der AUTOSAR-Spezifikation [AUT]. Die damit modellierten Software-Komponenten kapseln Teilfunktionalitäten oder sogar die vollständige Funktionalität eines Steuergeräts oder Steuergerätenetzwerkes. Um mit anderen Komponenten zu kommunizieren, besitzt jede Komponente Ports. Die Ports verschiedener Komponenten können miteinander verbunden werden, wenn die an den Ports angebotenen Schnittstellen kompatibel sind. Kompatibel in diesem Zusammenhang bedeutet, dass die zur Verfügung gestellten Daten diejenigen sind, die ein verbundenes Interface benötigt. In Abbildung 1 ist ein Beispiel der Software-Architektur einer Diebstahlwarnanlage gegeben.

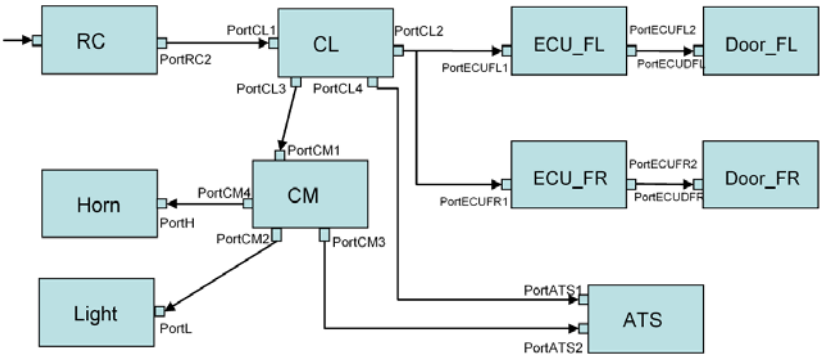


Abbildung 1: Software-Architektur einer Diebstahlwarnanlage

Das Verhalten einer einzelnen Komponente wird in [GH+06] mit Hilfe eines „Automotive Specific Automata“ spezifiziert (siehe Abbildung 2). Dies sind einfache Automaten die um Zeitannotationen erweitert wurden, wobei die Semantik über eine Abbildung auf Timed Automata [AD94] definiert ist. In Abbildung 2 ist ein „Automotive Specific Automata“ modelliert, der die drei Zustände *running, shutting_down* und *not_running* besitzt. Die Verweildauer in den Zuständen kann durch annotierte Zeitintervalle spezifiziert werden und die Aktivierung von Transitionen wird durch Nachrichten bzw. Guards modelliert.

Die formale Abbildung auf Timed Automata wird bei der Verifikation ausgenutzt, die mittels des Modelcheckers UPPAAL[1] durchgeführt wird. UPPAAL ist ein expliziter Modelchecker der als Eingabemodelle Timed Automata verwendet. Er ist frei verfügbar und wird im s-lab bereits seit einiger Zeit erfolgreich eingesetzt.

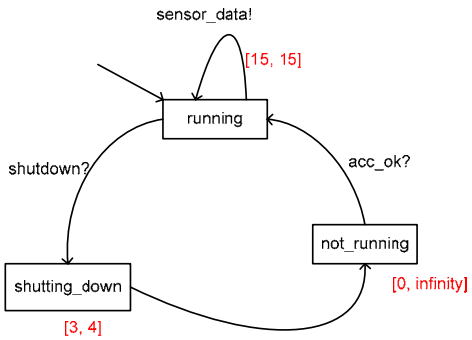


Abbildung 2:Beispiel eines Automotive Specific Automata

[1] http://www.uppaal.com

Für die Verifikation der Eigenschaft „Wie lange braucht das System, bis die Nachricht speed_ok gesendet wird“ muss diese Anfrage nun entsprechend formalisiert werden, damit die Verifizierung mit Hilfe des Modelcheckers UPPAAL möglich ist. Die Mächtigkeit der in UPPAAL verwendeten TCTL reicht hierfür jedoch nicht aus. Der Ansatz, das Verhalten aller Automaten global in einem Automaten zusammenzufassen und nur eine globale Uhr zu verwenden, ist nicht sinnvoll, da der dann gebildete Produktautomat ein Verhalten beschreibt, welches durch entstehende Seiteneffekte möglicherweise nicht mehr mit dem spezifizierten Verhalten einzelner Automaten übereinstimmt.

Um dieses Problem zu lösen wurde das Prinzip der szenariobasierten Verifikation angewandt [GH04, KM+02]. Hierbei wird für eine zu verifizierende Eigenschaft ein so genannter „Master“-Automat formuliert, welcher parallel zu den eigentlichen Automaten ausgeführt wird. Ein „Master“-Automat observiert das Verhalten der anderen Automaten (Timed Automata), wodurch dieser auf Einhaltung bestimmter Eigenschaften (durch temporallogische Formeln spezifiziert) überprüft werden kann. Da der „Master“-Automat genau das minimal benötige Verhalten aller vernetzten Komponenten für die zu überprüfende Eigenschaft in einem Automaten zusammenfasst, ist hierdurch eine korrekte, effiziente Verifikation möglich.

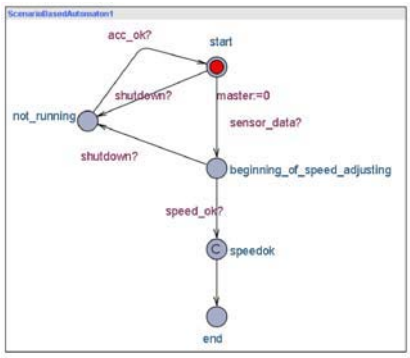


Abbildung 3: Beispiel für einen „Master-Automaten“

In Abbildung 3 ist ein Beispiel für einen „Master“-Automaten gegeben. Dieser beinhaltet z.B. den Zustand *not_running*, welcher dem Zustand *not_running* aus Abbildung 2 entspricht. Die anderen Zustände stammen aus anderen Timed Automata bzw. wurden explizit hinzugefügt.

3. Verifikation zeitlicher Anforderungen

Die manuelle Konstruktion sowohl der „Master“-Automaten, als auch der temporallogischen Formeln, ist sehr aufwändig und fehleranfällig. In beiden Fällen wäre es daher hilfreich einen entsprechenden Formalismus zu haben, der sowohl den „Master“-Automaten, als auch die temporallogischen Formeln automatisch generiert.

Die grundlegende Idee eine solche Generierung umzusetzen besteht darin, die verschiedenen Zeit-Anforderungen zu typisieren. Für jeden Typ ist es dann möglich, entsprechende Generierungsvorschriften zu definieren. In diesem Kapitel wird eine Typisierung der Zeit-Anforderungen für den Bereich der Karosserieelektronik im Bereich der Automobilindustrie vorgenommen. Als Grundlage für die Analyse der Zeit-Anforderungen werden Informationen verwendet, die aus den dort erstellten Pflichtenheften hervorgehen. Diese Zeit-Anforderungen kann man, abstrahiert betrachtet, auf die Zeitdauer zwischen Ereignissen beziehen. Es geht also immer um Ereignisse, ihr Auftreten, und die Zeitspanne zwischen ihrem Auftreten.

3.1 Typisierung der Zeit-Anforderungen

Die Zeit-Anforderungen für den Bereich Karosserieelektronik lassen sich in vier verschiedene Typen einteilen, wobei diese hierarchisch organisiert sind. Die vier Typen sind: „Maximale Verarbeitungszeit“, „Exakte Ausführungszeit“, „Synchronizität“ und „Konditionale Anforderungen“.

Maximale Verarbeitungszeit

Die maximale Verarbeitungszeit beschreibt, wie viel Zeit maximal zwischen zwei geordneten Ereignissen vergehen darf. Das folgende Beispiel verdeutlicht dies:

„Die Verarbeitungszeit zwischen Erkennen einer Bremslichtanforderung durch ein Lichtsteuergerät und dem Ansteuern der Ausgänge für die Bremsleuchten darf höchstens 50 ms betragen.“

Die maximale Verarbeitungszeit ist wie folgt definiert:

Eingabe: Start-Ereignis E_1
End-Ereignis E_2 wobei E_1 zeitlich vor E_2 eintritt
Zeit t_{max}

Bedingung: Die Zeit t zwischen dem Auftreten von E_1 und E_2 darf nicht größer sein als ein vorgegebener Wert t_{max}, also $t <= t_{max}$.

Exakte Ausführungszeit

Die exakte Ausführungszeit beschreibt, dass ein bestimmtes Ereignis genau nach einer festgelegten Zeit eintreten muss. Zusätzlich kann eine zeitliche Toleranz angegeben werden. Das folgende Beispiel verdeutlicht dies:

„Die Zentralverriegelung muss das Heckschloss bei Anforderung für eine Zeit von 600 ms ansteuern.“

Die Exakte Ausführungszeit ist wie folgt definiert:

Eingabe: Start-Ereignis E_1

End-Ereignis E_2 wobei E_1 zeitlich vor E_2 eintritt
Zeit t_{exakt}
Toleranz $t_{tol} >= 0$

Bedingung: Die Zeit t zwischen dem Eintreten von E_1 und dem Eintreten von E_2 muss sich im Intervall $[t_{min}, t_{max}]$ befinden, wobei $t_{min}=t_{exakt}-t_{tol}$ und $t_{max}=t_{exakt}+t_{tol}$ ist.

Sychronizität

Die Synchronizität beschreibt, dass mehrere Ereignisse ungeordnet innerhalb eines festgelegten Zeitintervalls eintreten müssen. Das folgende Beispiel verdeutlicht dies:

„Alle Leuchten (einschließlich Kontrolllampen), die gemeinsam in Betrieb sind, müssen gleichzeitig ein- bzw. ausgeschaltet werden. Es darf zu keinem vom Betrachter wahrnehmbaren Zeitversatz zwischen den einzelnen Lampen kommen."

Die Synchronizität ist wie folgt definiert:

Eingabe: Menge von Ereignisse $\{E_1, ..., E_n\}$
End-Ereignis E_2 wobei E_1 zeitlich vor E_2 eintritt
Zeit $t_{synchron}$

Bedingung: für alle i, j aus $\{1, ..., n\}$ gilt: $|t_i-t_j|<=t_{synchron}$ mit t_i ist Zeitpunkt des Eintretens von E_i

Konditionale Anforderungen

Konditionale Anforderungen beschreiben laufzeitabhängige Szenarien. Hier kommt es darauf an, was während der Laufzeit passiert, wann welches Ereignis eintritt. Das folgende Beispiel verdeutlicht dies:

„Werden von dem gleichen Schließzylinder innerhalb einer parametrierbaren Zeit zwei Entriegelungsbefehle abgegeben, wird das Fahrzeug komplett entriegelt."

Die konditionale Anforderung ist wie folgt definiert:

Eingabe: Start-Ereignis E_1
Bedingungs-Ereignis E_2 wobei E_1 zeitlich vor E_2 eintritt
Folge-Ereignis E_3
Zeit t_{cond}

Bedingung: Falls $t_2-t_1<=t_{cond}$ dann muss E_3 nach Ablauf der Zeit t_{cond} eingetreten sein; t_i ist der Zeitpunkt des Eintretens von E_i

Diese vier hier vorgestellten Typen sind ausreichend, um den Großteil der Zeit-Anforderungen automotiver Softwaresysteme zu beschreiben.

3.3 Generierung des „Master“-Automaten und der temporallogischen Formeln

Für jeden Anforderungstyp existiert eine Vorlage, die beschreibt, wie der jeweilige „Master“-Automat erstellt wird. Der Anwender muss lediglich den Typ der Anforderung und die Eingabedaten, wie zum Beispiel die Start- und Endereignisse, spezifizieren.

Im Folgenden ist beispielhaft für den Typ „Maximale Verarbeitungszeit“ und der Eigenschaft, dass 5ms nachdem die Fernbedienung betätigt wurde die Zentralverriegelung eingeschaltet ist, dargestellt, wie der „Master“-Automat generiert wird.Das Startereignis E_1 sei das Erkennen des Betätigens der Fernbedienung (`Remote_Control_portRC2_lock`), das Endereignis E_2 sei der Befehl zum Schärfen der Diebstahlwarnanlage (`Central_Locking_portCL4_ATSSharpening`) und die Zeit t_{max} sei 5 ms. In den Zuständen *Start* und *Waiting* wird nacheinander auf die entsprechenden Nachrichten gewartet. Der „Master“-Automat ist auf der rechten Seite der Abbildung 3 dargestellt.

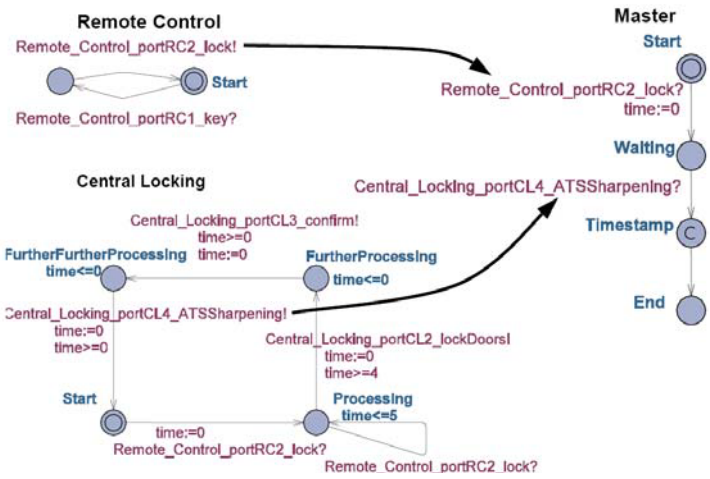


Abbildung 3: Generierung eines "Master"-Automaten

Nachdem der „Master“-Automat generiert wurde, muss nun noch die temporallogische Formel (in diesem Fall TCTL) generiert werden. Hierfür benötigt man lediglich die entsprechende Zeitangabe. Im obigen Beispiel sind dies t_{max} = 5 ms. Die folgende TCTL-Formel würde für das Beispiel wie folgt aussehen:

```
E<>Master.End
Master.Timestamp ~> Master.time <= 5
```

Der verwendete Modelchecker UPPAAL kann nun prüfen, ob der letzte Zustand immer erreicht wird und ob die Zeit zwischen den Ereignissen immer maximal 5 ms beträgt.

Äquivalent zur „Maximalen Verarbeitungszeit“ lassen sich auch für die anderen Anforderungstypen entsprechende Vorlagen zur automatischen Generierung des Master-Automaten angeben.

4. Zusammenfassung und Ausblick

Die Verifikation automotiver Software-Architekturen bereits in frühen Phasen der Software-Entwicklung wird in Zukunft immer mehr an Bedeutung gewinnen. Die Verifikation von Echtzeitaspekten spielt dabei eine besondere Rolle, da zeitliches Fehlverhalten von zusammenhängenden Funktionen ansonsten erst nach der Integration der einzelnen Funktionsimplementierungen erkannt werden kann. Dieses Fehlverhalten kann unter Umständen zu einem Redesign der gesamten Software führen und hat somit entschiedenen Einfluss auf Projektkosten und –laufzeiten.

In diesem Artikel wurde gezeigt, wie die Verifikation von Zeit-Anforderungen eines auf Komponenten basierenden Softwaresystems erfolgen kann. Dazu wird das zeitliche Verhalten der einzelnen Komponenten unabhängig voneinander mittels „Timed Automata“ bzw. einer Spezialisierung namens „Automotive Specific Automata“ beschrieben. Aus den einzelnen Timed Automata der Komponenten wird dann ein „Master“-Automat generiert. Dieser „Master“-Automat wird verwendet, um die Zeit-Anforderungen mittels temporallogischer Formeln komponentenübergreifend zu verifizieren. Sowohl der „Master“-Automat als auch die temporallogische Formel werden dabei automatisch generiert.

Bisher nicht berücksichtigt wurde die Verteilung der Software-Komponenten auf unterschiedliche Steuergeräte und die damit zusammenhängenden Laufzeiten der Netzwerkkommunikation oder aber Schedulingeffekte des Betriebssystems. Die Verifikation kann daher schon in frühen Phasen des Entwicklungsprozesses, zum Beispiel beim Fahrzeughersteller, eingesetzt werden. Sollen in späteren Phasen auch die o.a. Effekte nachgebildet werden, müssen entsprechend komplexere Automaten oder zusätzliche Automaten generiert werden, die z.B. die Laufzeiten von Nachrichten auf den Bussen berücksichtigen. Die Spezifikation des zeitlichen Verhaltens der einzelnen Komponenten selbst bleibt davon aber weiterhin unabhängig.

Eine weitere mögliche Erweiterung des Ansatzes besteht darin, die zu prüfenden Zeit-Anforderungen nicht in den temporallogischen Formeln (TCTL) zu codieren, sondern diese direkt in den „Master–Automaten“ als Invarianten zu annotieren. Der Vorteil hierbei wäre, dass die TCTL-Formeln deutlich kleiner werden und damit die Hoffnung besteht, das Modelchecking effizienter zu gestalten.

Acknowledgement

Für ihre Mitarbeit möchten wir Frau Petra Nawratil und Frau Renate Ristov herzlich danken. Sie haben mit ihren Diplom- bzw. Studienarbeiten entscheidende Beiträge für dieses Papier geliefert. Darüber hinaus bedanken wir uns bei den Mitarbeitern der Firma dSPACE GmbH und der Hella KGaA für Ihre tatkräftige Unterstützung.

Literaturverzeichnis

[AUT] http://www.autosar.org

[AD94] R. Alur and D. Dill. A theory of timed automata. In Theoretical Computer Science, 126(2): 183-235, 1994.

[GH04] S. Graf and J. Hooman. Correct Development of Embedded Systems. In Flavio Oquendo, Brian Warboys, and Ron Morrision, editors, *Proc. of the First European Workshop on Software Architecture, EWSA2004*, volume 3047 of *Lecture Notes in Computer Science*, pages 241-249, St Andrews, UK, May 21-22 2004, Springer Verlag.

[GH+06] M. Gehrke, M. Hirsch, P. Nawratil, O. Niggemann, and W. Schäfer: Scenario-Based Verification of Automotive Software Systems: In *Proc. 2nd Workshop on Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*, *Schloss Dagstuhl, Germany* (Holger Giese, Bernhard Rumpe, and Bernhard Schätz, eds.), volume Informatik-Bericht 2006-01, Technische Universität Braunschweig, pages 35-42, January 2006.

[KM+02] A. Knapp, S. Merz, and C. Rauh. Model Checking timed UML State Machines and Collaborations. 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), Oldenburg, September 2002, Lecture Notes in Computer Science, volume 2469, pages 395-414, Springer-Verlag, 2002.

Das MATE Projekt – visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen

Ingo Stürmer[1], Heiko Dörr[2], Holger Giese[3], Udo Kelter[4], Andy Schürr[5], Albert Zündorf[6],

[1]Model Engineering Solutions, stuermer@acm.org
[2]DaimlerChrysler AG, heiko.doerr@daimlerchrysler.com
[3]Universität Paderborn, hg@uni-paderborn.de*
[4]Universität Siegen, kelter@informatik.uni-siegen.de
[5]TU Darmstadt, andy.schuerr@es.tu-darmstadt.de
[6]Universität Kassel, zuendorf@uni-kassel.de

Abstract: Die modellbasierte Entwicklung beginnt sich als Standardparadigma in der Steuergerätesoftwareentwicklung zu etablieren. Um die Wirksamkeit und Effizienz der modellbasierten Entwicklung zu erhöhen, sind Richtlinien für die Modellierung unerlässlich. Diese manuell zu überprüfen ist aufwändig und fehleranfällig. Das Projekt MATE (MATLAB Simulink/Stateflow Analysis and Transformation Environment) hat sich deshalb zum Ziel gesetzt, die bislang meist manuelle Prüfung der Einhaltung bzw. Verletzung von Modellierungsrichtlinien zu automatisieren, sowie vor allem Modelltransformationen zur Korrektur entdeckter Mängel und zur Unterstützung von Entwicklungsschritten und die Visualisierung der Unterschiede verschiedener Entwicklungsstände zu untersuchen.

1 Einleitung

Kennzeichnend für die modellbasierte Entwicklung eingebetteter Software im Automobil ist die frühzeitige Beschreibung der Regelungs- und Steuerungsalgorithmen durch ausführbare Modelle unter Verwendung von Blockschaltbildern und Zustandsübergangsdiagrammen. Hierbei wird der Ingenieur unterstützt durch Modellierungs- und Simulationswerkzeuge wie *MATLAB Simulink/Stateflow* [MW06]. Die zunehmende Komplexität der elektronischen Systeme im Kraftfahrzeug schlägt sich entsprechend in den Modellen nieder. Modellierungswerkzeuge bieten jedoch nur bedingt Mechanismen zur Beherrschung dieser Komplexität und zur Unterstützung der Entwickler. Besonders deutlich zeigen sich diese Probleme an (1) der bisher nicht hinreichend automatisierten Unterstützung der Überprüfung dutzender von Modellierungsrichtlinien und der Korrektur des Modells entsprechend der Regeln, (2) der mangelhaften Versionierungsunterstützung und der Darstellung von Unterschieden zwischen Modellen, und (3) der immer noch unzureichenden Integration externer Werkzeuge, z.B. Werkzeuge des Requirements Engineering, mit der Modellierungsumgebung (etwa in Punkten wie der bidirektionalen Propagation von Änderungen und der Verwaltung von Konsistenzbeziehungen zwischen Entwicklungsartefakten, die in verschiedenen Werkzeugen gespeichert sind).

---

* Augenblicklich Lehrstuhlvertreter am Hasso-Plattner-Institut der Universität Potsdam

Im Rahmen des Projekts *MATE* (*M*ATLAB Simulink/Stateflow *A*nalysis and *T*ransformation *E*nvironment) untersuchen sechs Projektpartner (siehe Autorenliste), wie mit Hilfe von Modellanalyse- und Transformationstechniken die oben genannten Probleme bei der modellbasierten Entwicklung gelöst werden können. Dabei spielen die Modellierungs- und Modelltransformationsumgebung *Fujaba* [Fuja06] sowie die darauf aufbauende Metamodellierungsumgebung *MOFLON* [MOF06] eine herausragende Rolle für die Spezifikation und Generierung von Zugriffschnittstellen, Modell-Repositories, Analysefunktionen und Editieroperationen für Matlab Simulink/Stateflow-Modellen. Fujaba und MOFLON bieten hierfür eine Vereinigung der (Meta-)Modellierungsstandards der OMG mit Graphtransformationen als präzise definiertem visuellem Spezifikationsansatz an.

Ziel des Projektes MATE ist es, eine eng mit den Werkzeugen MATLAB Simulink und Stateflow sowie dem darauf aufbauenden *Simulink ModelAdvisor* [MW06] integrierte Unterstützung folgender Entwicklungsaktivitäten (Anwendungsszenarien) anzubieten:

1. die Überprüfung von Modellierungsrichtlinien (durch Suche nach Anti-Patterns)
2. die Berechnung von Metriken (für Identifikation zu überarbeitender Modellteile)
3. die Generierung von Reparaturvorschlägen für Richtlinienverletzungen
4. die batchorientierte Durchführung vorgeschlagener Reparaturen von Richtlinienverletzungen durch Modelltransformationen
5. die Unterstützung komplexer interaktiver Editieroperationen (Einsatz empfohlener Entwurfsmuster, interaktive Elimination von Anti-Entwurfsmustern etc.)
6. sowie die Berechnung und Visualisierung der Unterschiede von Modellversionen

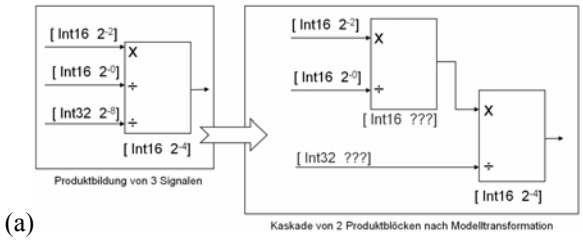
Die in der Praxis ebenfalls erwünschte Integration von MATLAB Simulink/Stateflow-Modellen mit anderen Entwicklungsartefakten (wie etwa mit in DOORS [Tel06] verwalteten Anforderungen) wurde im MATE-Projekt zunächst ausgeklammert, da die Integration der Daten verschiedener Werkzeuge sowie die Verwaltung bidirektionaler Traceability-Links Schwerpunkt des separaten Projekts *ToolNet* [ADS02] ist, das von einer Teilmenge der MATE-Partner durchgeführt wird.

Das Papier ist im Weiteren wie folgt gegliedert: In Abschnitt 2 wird zunächst die Spezifikation und Erkennung von Mustern erläutert, die die Basis für die Berechnung von Modellmetriken, die Überprüfung von Modellierungsrichtlinien und die Elimination von Richtlinienverletzungen in MATE bildet. Darauf aufbauend wird in Abschnitt 3 beschrieben, wie alle Arten von Änderungsoperationen auf MATLAB Simulink/Stateflow-Modellen in Form von Graphtransformationen spezifiziert werden. Die Realisierung der MATE-Werkzeugumgebung wird dann in Abschnitt 4 knapp skizziert. Den Abschluss des Beitrags bilden ein Vergleich mit verwandten Arbeiten in Abschnitt 5 und die üblichen Schlussbemerkungen in Abschnitt 6.

Damit werden von den oben aufgeführten und von MATE unterstützten Anwendungsszenarien die Spezifikation von Metriken sowie die Berechnung und Visualisierung von Modelldifferenzen in diesem Beitrag nicht ausführlich vorgestellt. Insbesondere bezüglich der Berechnung der Unterschiede von Modelldifferenzen sei der Leser deshalb auf weiterführende Darstellungen und Literatur wie [KWN05, SiDiff] hingewiesen.

2 Mustererkennung

Es hat sich gezeigt, dass die Einhaltung von Modellierungsrichtlinien und die Verwendung standardisierter Muster (Pattern) für die erfolgreiche Umsetzung von Anforderungen in ausführbare Modelle unerlässlich sind [CD+05]. Die schiere Anzahl einzuhaltender Regeln (derzeitig 200 bei DaimlerChrysler) verlangt jedoch deren automatisierte Prüfung am Modell. Hierfür existieren bereits Werkzeuge, die auf Basis von MATLAB M-Scripts Regelverletzungen aufdecken, wie z.B. der *Simulink Model Advisor* [MW06] oder *MINT* [Ric06]. Bei der Suche nach Modellierungsmustern bzw. Regelverletzungen, die auf zu vermeidenden Mustern basieren, zeigen diese Skripte aber deutliche Nachteile. Die Programmierung der Mustersuche ist aufwändig; die Skripte sind nur von Experten zu verstehen und schwer zu warten (siehe etwa Abb. 7 am Ende des Beitrags).



(a) (b)

Abbildung 1: Problem bei der automatisierten Transformation eines Product-Blocks

Am folgenden Beispiel soll erläutert werden, welche Probleme bei der Anwendung von Modellierungsrichtlinien entstehen, wenn Modelle entsprechend dieser Regeln automatisiert (a) analysiert und (b) modifiziert werden sollen. Arithmetische Operationen, wie z.B. die Produktbildung zweier oder mehrerer Operanden, werden in Simulink mit Hilfe funktionaler Blockschaltbilder realisiert. Abb. 1 (links) zeigt einen so genannten Product-Block, mit drei Eingängen, deren Festkomma-Datentypen (int16, int32) auf unterschiedliche Genauigkeiten skaliert sind (2^{-0}, 2^{-2}, 2^{-8}). Das Ergebnis der Produktbildung wird an den Blockausgang des Product-Blocks propagiert. Diese Art der Modellierung ist problematisch, wenn das Modell mittels eines Codegenerators in Festkomma-Code übersetzt werden soll. Für die Übersetzung eines Product-Blocks mit mehr als zwei Operanden werden Skalierungsinformationen für Zwischenergebnisse benötigt (siehe Abb. 1, rechts), die der Codegenerator nicht automatisiert berechnen kann und die in Abb. 1 (a) auch nicht vom Entwickler angegeben werden können. Daher werden Product-Blöcke mit mehr als zwei Signaleingängen für die Seriencodegenerierung verboten.

Es erscheint sinnvoll, Entwickler durch die automatisierte Aufdeckung und halbautomatische Elimination solcher Richtlinienverletzungen zu unterstützen. Für die Entwicklung entsprechender Überprüfungen eignen sich Werkzeuge, die auf Graphtransformationen basieren, wie sie etwa vom Open-Source-Projekt *Fujaba* [Fuja06] mit dem Metamodellierungs-Plugin *MOFLON* [MOF06] angeboten werden. *Fujaba/MOFLON* erlaubt die visuelle Spezifikation verbotener Muster sowie deren Ersetzung durch empfohlene Entwurfsmuster auf hohem Abstraktionsniveau bei gleichzeitiger Unterstützung der OMG-Metamodellierungs-Standards zur modellbasierten Softwareentwicklung (MOF 2.0).

Im Folgenden werden wir zeigen, wie aufbauend auf einem Prototypen [GM+06] im Rahmen des MATE-Projekts die oben skizzierten Ideen exemplarisch umgesetzt wurden. Es wurde dafür zunächst in Kooperation mit dem artverwandten MESA-Projekt [FHR06] ein Metamodell für alle Modellierungselemente von MATLAB Simulink/Stateflow als MOF 2.0 (und damit auch als UML 2.0) Klassendiagramm entwickelt. Zur Laufzeit wird der Zugang zu Instanzen dieses Metamodells auf zwei verschiedene Arten unterstützt: (1) mittels Lese- und Schreibzugriffen über einen *Adapter* direkt auf die in Matlab Simulink/Stateflow gespeicherten Modelle und (2) durch Export der zu untersuchenden Modelle in ein externes *Repository* (mit anschließendem Re-Import) mit einer entsprechenden Schnittstelle für Lese- und Schreibzugriffe (siehe auch Abschnitt 5). Aufbauend auf dem Metamodell werden gemäß der Modellierungsrichtlinien zu suchende Muster mit Hilfe von *Analyseregeln* beschrieben, die dann mittels eines Musterinterferenzmechanismus von Fujaba [NS+02] in einem MATLAB Simulink- oder Stateflow-Modell identifiziert werden. Nachfolgend können dann *Transformationsregeln* in Form von Story-Diagrammen (siehe Abschnitt 3 und [NNZ00]) dazu genutzt werden, die ggf. vorhandenen Regelverletzungen durch Modelltransformationen zu korrigieren.

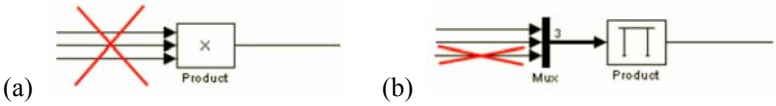


Abbildung 2: Betrachtete Fälle zur Ableitung von Analyseregeln

Ein erster notwendiger Schritt, um die in Abb. 1 angedeutete automatische Transformation zu ermöglichen, ist die Identifikation von verbotenen Situationen (siehe Abb. 2) mittels Analyseregeln. So darf, wie in Abb. 2 (a) angedeutet, kein Product-Block mit mehr als zwei Eingängen vorhanden sein oder, wie in Abb. 2 (b) gezeigt, kein vektorwertiger Eingang mit mehr als zwei Elementen existieren.

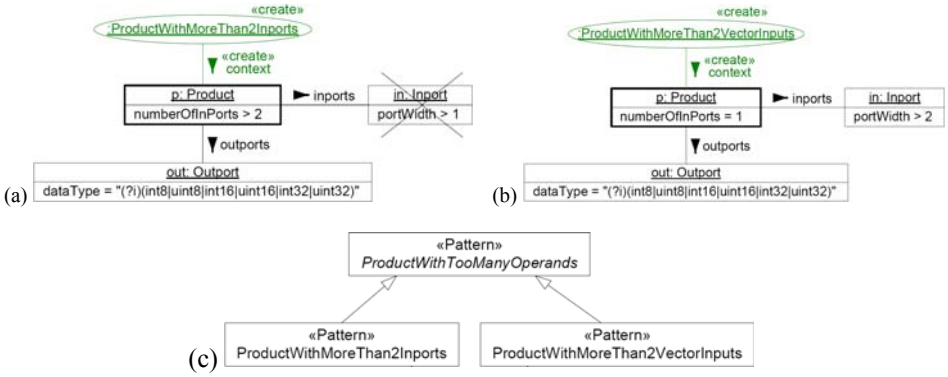


Abbildung 3: Analyseregeln für Produktblöcke mit zu vielen Eingängen

In Abb. 3 (a) und (b) sind zwei verschiedene Regeln dargestellt, die zur Überprüfung der Modellierungsrichtlinie aus Abb. 1 auf Basis des erstellten Metamodells dienen. In der Regel aus Abb. 3 (a) wird dabei mittels des Metamodell-Attributs numberOfInPorts spezifiziert, dass ein Produktblock (p:Product) mehr als zwei Eingänge hat, und dass der Da-

tentyp des Ausgangsports einem der relevanten Festkommaformate entspricht. Wird eine solche Situation gefunden, so wurde eine Verletzung der Modellierungsrichtlinie aus Abb. 2 (a) erkannt und die Situation wird durch eine entsprechende Annotation vom Typ ProductWithMoreThan2Inports am Product-Block kenntlich gemacht. Um auszuschließen, dass die Erkennung und anschließende Problembehandlung fälschlicherweise auch auf Blöcken mit vektorwertigen Inports ausgeführt wird, ist dies durch einen negativen vektorwertigen Inport-Knoten (ausgekreuzt) ausgeschlossen. Die zweite Analyseregel in Abb. 3 (b), die den Fall aus Abb. 2 (b) betrachtet, verbietet einen vektorwertigen Inport mit mehr als 2 Elementen. Beide Fälle lassen sich, wie in Abb. 3 (c) beschrieben, zu einem allgemeineren Muster namens ProductWithTooManyOperands zusammenfassen. Diese Verallgemeinerung trifft immer dann auf einen betrachteten Modellausschnitt zu, wenn eine ihre Spezialisierungen zu dem betrachteten Ausschnitt passt.

3 Modelltransformation

Neben der Erkennung von Regelverletzungen können *unidirektionale Transformationsregeln* in Form von Story-Diagrammen dazu genutzt werden, an die Analyseregel anknüpfende Modelltransformationen zu realisieren. So kann die Identifikation der entsprechenden Instanzsituation im Modell durch die vorgestellten Regeln aus Abb. 3 als Ansatzpunkt dazu genutzt werden, den Fehler durch eine Modelltransformation zu beheben. Dazu werden einfach weitere Elemente, die für eine Transformation notwendig sind, wie z.B. die über die Links verbundenen Blöcke, entsprechend eingeblendet.

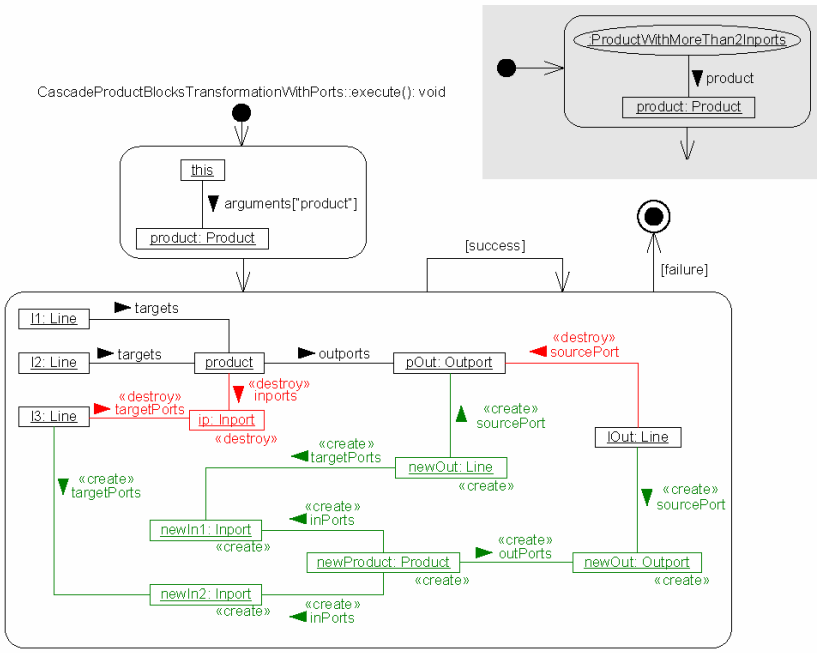


Abbildung 4: Transformationsregel für Blöcke mit mehr als zwei Eingängen

Die entsprechende Transformationsregel in Abb. 4 setzt auf der Erkennung der Situation auf und beschreibt, welche Veränderungen im Modell notwendig sind, damit die Modellierungsrichtlinie eingehalten wird. Dabei wird in der ersten Aktivität des dazu verwendeten erweiterten UML-Aktivitätsdiagramms die zuvor gefundene Situation durch Zugriff über eine qualifizierte Assoziation gebunden. Wie rechts oben in Abb. 4 angedeutet, soll dieses Binden durch entsprechend an der Musterdefinition orientierte Visualisierung zukünftig noch eingängiger gestaltet werden, indem man die in den Analyseregeln aus Abb. 3 verwendeten Annotationen direkt einblendet. Sobald die entsprechenden Elemente gebunden sind, wird im Aktivitätsdiagramm die zweite Aktivität angestoßen. Hier erlauben die in die Aktivität visuell eingebetteten Story Pattern die benötigte Transformation mittels zu löschender («destroy») oder zu erzeugender («create») Modellelemente zu spezifizieren. Durch eine Schleife im Aktivitätsdiagramm wird dabei beschrieben, dass dieser atomare Transformationschritt solange angewendet wird, bis keine Anwendungsstelle mehr gefunden werden kann.

An den gezeigten Beispielregeln kann man erkennen, wie die für die Verletzung von Modellierungsrichtlinien charakteristischen Situationen systematisch durch ausschnittartige Instanzsituationen des Metamodells und zusätzliche Bedingungen in Form der vorgestellten Regeln spezifiziert werden können.

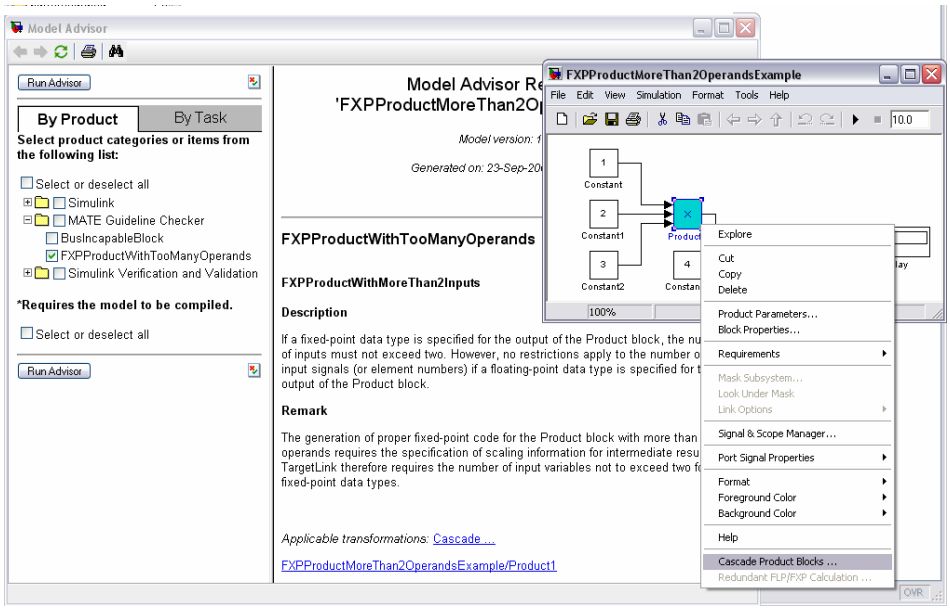


Abbildung 5: Analyseregeln und Transformationsregeln im ModelAdvisor

In Abb. 5 ist die Integration der Analyse- und Transformationsregeln in den Simulink ModelAdvisor angedeutet. Die Analyseregeln werden als normale Konsistenzregeln angezeigt und können entsprechend vom ModelAdvisor angestoßen werden. In diesem Fall ist dies die Regel ProductWithTooManyOperands.

Ist eine Situation gefunden worden, bei der die Regel verletzt wird, so wird durch den ModelAdvisor die entsprechende Stelle per Link zugänglich gemacht und ein die entsprechende Regel erläuternder Text angezeigt. Gibt es anwendbare Transformationsregeln, die zu einer Korrektur der Situation führen würden, so werden diese nun ebenfalls angezeigt. So wird in unserem Beispiel die Regel CascadeProductBlocksTransformationWithPorts dargestellt. Daneben ist, wie in Abb. 5 gezeigt, aber auch eine Auswahl derselben Regel für die selektierte Situation aus dem Kontextmenü in Matlab/Simulink möglich.

4 MATE Architektur

Die in Abb. 6 dargestellte Architektur von MATE bildet das Grundgerüst für alle skizzierten Anwendungsszenarien. Die Basis des Gesamtsystems bildet das Modellierungswerkzeug MATLAB Simulink/Stateflow mit der Erweiterung um den ModelAdvisor, der bereits ein interaktives Rahmenwerk für die Verwaltung und Ausführung von Analyseregeln sowie die Betrachtung von Analyseergebnissen auf *einem* Modell bereitstellt. Damit integriert wurde ein Visualisierungswerkzeug, das auf einigen im ToolNet-Projekt [ADS02] entwickelten Bausteinen zur Kommunikation mit MATLAB Simulink und Stateflow aufsetzt und beispielsweise die Markierung beliebiger Modellierungselemente drastisch vereinfacht. Dabei wird vor allem die gleichzeitige Betrachtung *mehrerer* Modellversionen und deren Unterschiede unterstützt.

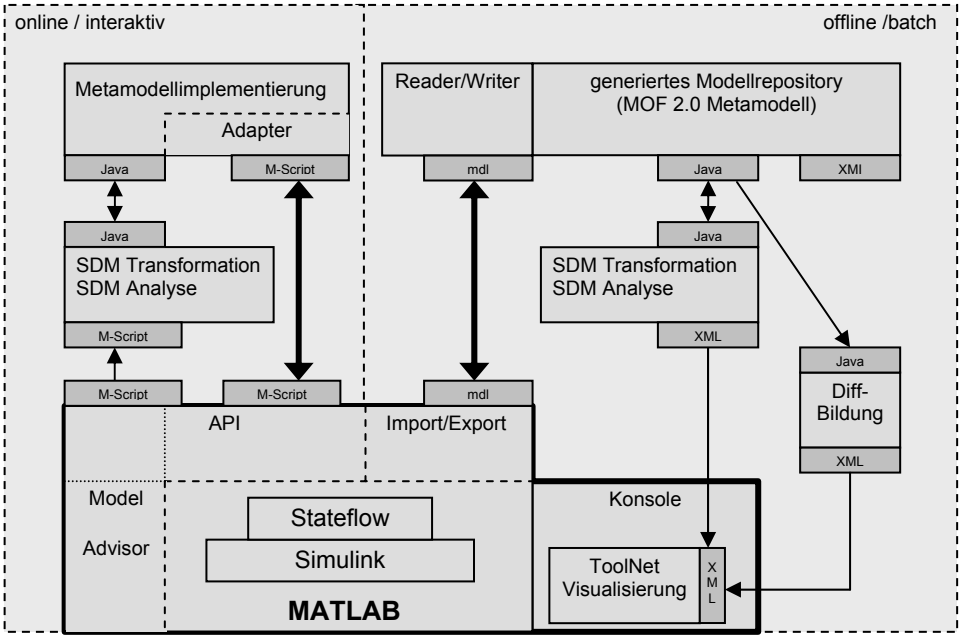


Abbildung 6: MATE-Architektur (vereinfacht)

Auf der beschriebenen Basisschicht setzen nun zwei weitgehend unabhängige Teilsysteme auf, die beide als Kernstück von Fujaba generierten Java-Code für Modell-Analysen und -Transformationen enthalten. Als Basis wird hierfür das in den Abschnitten 2 und 3 vorgestellte *S*tory *D*riven *M*odeling (SDM) eingesetzt.

Die *Online*-Lösung unterstützt zunächst die durch den ModelAdvisor gesteuerte interaktive Analyse von Modellen. Dabei ruft der ModelAdvisor *M-Script-Fragmente*[†] auf, die ihrerseits Java-Methoden aktivieren. Diese Methoden unterstützen die Identifikation empfohlener Design- oder verbotener Anti-Pattern in den untersuchten Modellen sowie die Ersetzung gefundener Anti-Pattern durch geeignete Design-Pattern. Sie kommunizieren dabei über proprietäre Schnittstellen mit einem Adapter, der alle auf Modellen denkbaren Lese- und Schreiboperationen unterstützt. Dieser Adapter greift wiederum über die M-Script-Schnittstelle auf die in MATLAB Simulink/Stateflow verwalteten Modelle zu. Die Analyseergebnisse werden auf umgekehrtem Weg zurückgegeben und im Model-Advisor angezeigt. Auf demselben Weg ist es möglich, Transformationen auf durch die Analyse als fehlerhaft gekennzeichneten oder anderweitig selektierten Modellstellen interaktiv durchzuführen.

Die *Offline*-Lösung geht bei der Analyse und Transformation von Simulink/Stateflow-Modellen anders vor, da hier nicht die interaktive Manipulation von Simulink/Stateflow-Modellen, sondern aufwändige Analysen im Mittelpunkt stehen, die den Aufbau zusätzlicher Index-Strukturen erfordern. Deshalb werden zunächst Simulink- und Stateflow-Modelle in ihrem propietären mdl-Format exportiert und in ein eigenständiges Modell-Repository importiert. Dieses Repository verfügt über eine Java-API, die den von Sun entwickelten Standard JMI unterstützt. Zur Datenhaltung kann deshalb entweder das Metadaten-Repository MDR von Sun oder bevorzugt eine von den Projektpartnern selbst entwickelte und deutlich effizientere Lösung verwendet werden, die alle Modellierungsdaten im Hauptspeicher hält. Letztere wird aus dem in Abschnitt 2 erwähnten MATLAB Simulink/Stateflow-Metamodell mit Hilfe des Fujaba-Plugins MOFLON generiert.

Auf dem generierten Repository werden die zur Verfügung stehenden Analyse- und Transformationsmethoden ausgeführt, die einerseits die im Repository abgelegten Modelle verändern und andererseits Analyseberichte in Form von XML-Dateien erzeugen können. Die veränderten Modelle können wiederum als mdl-Dateien in die MATLAB Simulink/Stateflow Umgebung eingelesen werden. Die XML-Analyseberichte werden von der bereits oben erwähnten ToolNet-Visualisierung weiterverarbeitet. Sie öffnet sowohl das analysierte Modell als auch das daraus hervorgegangene korrigierte Modell und zeigt jeweils zueinander korrespondierende fehlerhafte Stellen im Ursprungsmodell und korrigierte Stellen im neuen Modell an. Die zur Analyse und Transformation eingesetzten Java-Methoden werden bei der Offline-Lösung ebenfalls visuell spezifiziert und mit Hilfe von Fujaba/MOFLON in Java-Code übersetzt. Ähnlich funktioniert die Berechnung von Modelldifferenzen, die zwei im Repository abgelegte Modelle miteinander vergleicht und deren Unterschiede mit dem selben Visualisierungsmechanismus wie die Analysen und Transformationen darstellt.

[†] M-Script ist die für die Manipulation von Simulink- und Stateflow-Modellen angebotene Script-Sprache.

6 Verwandte Arbeiten

Es gibt bereits eine Handvoll von Ansätzen wie den Simulink Model Advisor [MW06] oder MINT [Ric06], die meist auf der Basis von MATLAB M-Scripts Regelverletzungen in Simulink- oder Stateflow-Modellen aufdecken. Wie an Abb. 7 ersichtlich (im Anhang der Veröffentlichung), ist die „Programmierung“ solcher Regeln jedoch erheblich komplexer und aufwändiger als die im MATE-Projekt realisierte Spezifikation von Analyseregeln und anschließende Generierung der ausführbaren Regeln. Dies liegt zum einen daran, dass die von uns verwendete Adapterschicht immer wiederkehrende Zugriffscodeblöcke geeignet kapselt und diese somit wiederverwendet werden. Darüber hinaus wird vor allem in den Analyse- und Transformationsregeln auf einem höheren Abstraktionsniveau spezifiziert, so dass aufwändige Navigationsoperationen auf dem Metamodell in den Regeln wesentlich kompakter beschreibbar sind.

Des Weiteren existieren Arbeiten im Rahmen des MESA-Projekts [FHR06], die ebenfalls auf einem höheren Abstraktionsniveau Modellierungsrichtlinien spezifizieren und daraus ausführbaren Code generieren. Ausgangspunkt ist auch dort das von uns mitverwendete MATLAB Simulink/Stateflow-Metamodell; allerdings wird anstelle von visuellen regelbasierten Spezifikationstechniken in MESA die textuelle logikbasierte Object Constraint Language (OCL) der OMG eingesetzt. Ein umfassender Vergleich von Fujaba SDM-Diagrammen einerseits und OCL Constraints andererseits in punkto Ausdrucksstärke, Erstellungsaufwand, Lesbarkeit und industrieller Akzeptanz liegt bislang noch nicht vor. Klar ist allerdings, dass SDM-Diagramme direkt die Spezifikation von Modelltransformationen unterstützen, während bei OCL zusätzliche Erweiterungen zu einer Transformationssprache benötigt werden. Darüber hinaus eignen sich SDM-Diagramme eher zur Beschreibung komplexer Entwurfsmuster, während OCL eher auf die Beschreibung logischer Formeln mit Konjunktionen und Implikationen zugeschnitten ist. So erscheint uns auf Dauer die Kombination von SDM und OCL sinnvoll, wie sie im Rahmen der Metamodellierungsumgebung Fujaba/MOFLON bereits in Angriff genommen wurde.

Abschließend sei darauf hingewiesen, dass keine der verwandten Arbeiten den Bogen von rein batchorientierten Analyseregeln hin zu interaktiven Modelltransformationen spannt, sodass nur MATE neben der Erkennung von Richtlinienverletzungen auch deren Behebung entweder im Batchbetrieb oder mit komplexen Editierkommandos unterstützt.

7 Zusammenfassung

Das Projekt MATE – eine Kooperation von DaimlerChrysler mit vier Universitäten und der Firma Model Engineering Solutions – stellt eine Architektur, Vorgehensweisen und Werkzeuge zur Verfügung, die die Lösung praxisrelevanter Probleme der modellbasierten Entwicklung mit MATLAB Simulink/Stateflow ermöglicht. Die Realisierung des Ansatzes wurde am Beispiel der automatisierten Überprüfung und Anwendung von Modellierungsrichtlinien und Modelltransformationen beschrieben, die mit Hilfe von Fujaba und dem dazugehörigen Metamodellierungs-Plugin MOFLON auf hohem Abstraktionsniveau spezifiziert und in ausführbaren Java-Code übersetzt werden.

Durch Anwendung weiterer in *Fujaba* verfügbarer Konzepte wie *bidirektionale Konsistenzregeln* [GW06] können auf Basis der im ToolNet-Projekt [ADS02] entwickelten Infrastruktur zur Integration von Werkzeugen auch darüber hinaus benötigte Aspekte wie die (halb-)automatische Überwachung von Konsistenzbeziehungen und Propagation von Änderungen zwischen MATLAB Simulink/Stateflow-Modellen einerseits und in DOORS verwalteten Anforderungen umgesetzt werden.

Darüber hinaus wurde die MATE-Architektur um SiDiff [KWN05, SiDiff], ein generisches Werkzeug zur Berechnung symmetrischer Differenzen zwischen zwei graphartig strukturierten Dokumenten erweitert. SiDiff selbst ist zu diesem Zweck auf den Vergleich von MATLAB Simulink-Modellen so konfiguriert worden, dass nur wesentliche Änderungen hervorgehoben werden und vor allem selbst umbenannte oder verschobene Modellierungselemente auf Basis entsprechender Heuristiken einander zugeordnet werden können. Die gefundenen Differenzen werden gegenwärtig durch paarweises Hervorheben korrespondierender Elemente in Matlab/Simulink visualisiert. Eine Erweiterung von SiDiff auf Stateflow-Modelle wird in Kürze zur Verfügung stehen.

Literatur

[ADS02] Altheide, F.; Dörr, H.; Schürr, A.: Requirements to a Framework for sustainable Integration of System Development Tools, in: *Proc. of 3rd European Systems Engineering Conference (EuSEC'02)*, Toulouse: AFIS PC Chairs (2002), 53-57

[CD+05] Conrad, M.; Dörr, H.; Fey, I.; Pohlheim, H.; Stürmer, I.: Guidelines und Reviews in der modellbasierten Entwicklung von Steuergeräte-Software, in: *Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik*, Expert-Verlag (2005)

[FHR06] Farkas T.; Hein, Ch.; Ritter, T.: Automatic Evaluation of Modelling Rules and Design, in: *Second Workshop "From code centric to model centric software engineering: Practices, Implications and ROI"*. Bilbao, Spain (2006)

[GW06] Giese, H.; R. Wagner: Incremental Model Synchronization with Triple Graph Grammars, in *Proc. of 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Genoa, Italy, LNCS 4199, Springer Verlag (2006), 543-557

[KWN05] Kelter, U.; Wehren, J.; Niere, J.: A Generic Difference Algorithm for UML Models, in: *Proceedings of the SE 2005*, Essen, Germany (2005)

[Fuja06] *Fujaba*, http://www.fujaba.de and ./projects/reengineering/index.html (2006)

[GM+06] Giese, H.; Meyer, M.; Wagner, R.: A Prototype for Guideline Checking and Model Transformations for MATLAB/Simulink, in: *Proc. of the Fujaba Days* (2006)

[MW06] The MathWorks: http://www.mathworks.com/products (2006)

[NNZ00] Nickel, U.; Niere, J.;. Zündorf, A.: Tool demonstration: The FUJABA environment, in *Proc. of the 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, ACM Press (2000), 742-745

[NS+02] Niere, J.; Schäfer, W.; Wadsack, J.; Wendehals, L.; Welsh, J.: Towards Pattern-Based Design Recovery, in *Proc. of the 24th International Conference on Software Engineering (ICSE)*, Orlando, Florida, USA, ACM Press (2002), 338-348

[Ric06] Ricardo, Inc.: *MINT*, http:/www.ricardo.com/mint (2006)

[SiDiff] Universität Siegen: SiDiff Difference Tool Set; www.sidiff.org (2006)

[Tel06] Telelogic, Inc.: *DOORS*, http://www.telelogic.com (2006)

[MOF06] Technische Universität Darmstadt: *MOFLON*, http://www.moflon.org (2006)

```
 1  function [ok, test_data] = mint_tl_0009(cmd_s, scope_list_st)
 2  % MINT_TL_0009 -- Limitations with regard to Operand Numbers for the Product Block
 3  %

...

10  %
11  % Copyright 2006 DaimlerChrysler AG
12  % (Written by I. Stuermer)
13  %

...

23  % Initialize the return variable.
24  ok=1;
25  % Registration data.
26  if strcmp(cmd_s,'register')
27  test_data.name = 'TL_0009 - Limitations with regard to Operand
                      Numbers for the Product Block';
28    %test_data.features = {'mint_dcag'};
29    test_data.number = 10009;
30    test_data.scope = 'subsystem';
31    test_data.category = 'Warning' ;
32    test_data.ispseudo = 0 ;
33    test_data.generated_scope = '' ;
34    test_data.data_gen = { ...
35      } ;
36    test_data.data_req = { ...
37        'subsys_handle'...
38      } ;
39
40    return ; %%%%%%% RETURN %%%%%%%
41  end

42  % Ticker update.
43  ok=mint_tdata_run_progress;
44  if(~ok)
45    return;
46  end

47  % Unpack subsystem handles
48  subsys_handle = mint_tdata_data_get(scope_list_st,...
49    'subsys_handle');
50  subsys_handle = [subsys_handle(:)];

51  top_h = mint_get_handle(mint_tdata_top_get);
52  model_name=get_param(top_h,'Name');

53  % Create list of erroneous product blocks inside controller
54  f_block_h = [];

55  % create list of all product block with fixed-point data types
56  all_block_h= [] ;

57  % We want to check the CompiledPortDataType of the Product outport
58  % In order to do so, we need to set the model into compile mode.
59  % Before, term compilation if model is already in compile mode
60  try
61    eval(sprintf('%s([],[],[],''term'')',model_name));
62  end

63  try
64    %switch to compile mode
65    eval(sprintf('%s([],[],[],''compile'')',model_name));
66
67  catch
68    disp('TL_0009 - Unable to set model into compile mode');
69    errstr = strrep(lasterr, 'Error using ==> eval', '');
70    stern = '*****************************************************';
71    fstr = sprintf('\n%s\n%s\n\n%s\n\n', stern, errstr, stern);
72    disp(fstr);
73  end

74  % get all product blocks
75  block_h = mint_find_system(top_h,...
76                'BlockType','Product');
77  all_block_h={all_block_h;block_h};

78  %loop over all product blocks
79  for k=1:length(all_block_h)
80
81    %get port handles of actual product block
82    porth= get_param(all_block_h(k),'PortHandles');
83
84    % check whether its outport data type is an integer (fixed-point)
85    % data type
86    outdata= get_param(porth.Outport,'CompiledPortDataType');

87    % prepare regular expression for query
88    regexp= '[int8|int16|int32|fix]';
89    res=regexpi(outdata, regexp, 'match');
90
91    %get the name of the actual block
92    block_name=get_param(all_block_h(k),'Name');
93
94    % if res is not empty, we found a product block whose outport data type
95    % is (scaled) integer
96    if ~isempty(res)
97
98      %disp(sprintf(' Output data type of block %s is %s', block_name, outdata));

99
100     % CHECK WHETHER PRODUCT BLOCK HAS MORE THAN TWO INPUTS
101     if (length(porth.Inport) > 2)
102      f_block_h = [f_block_h;all_block_h(k)];
103     else
104       % So we got only two inports!
105
106       % CHECK WHETHER PRODUCT BLOCK IS DRIVEN BY A VECTOR
107       % We have to consider two cases
108       % (1) one inport with vector > 2
109       % (2) two inports   (a) first inport with vector >=2, or
110       %                   (b) second inport with vector >= 2
111
112       % CASE 1
113       if (length(porth.Inport)==1)
114         indata= get_param(porth.Inport(1),'CompiledPortDimensions');
115         vector_size= indata(2);
116         if ((vector_size) > 2)
117          f_block_h = [f_block_h;all_block_h(k)];
118           %disp(sprintf(' Input dimension of block %s inport 1 is %s',
                              block_name, vector_size));
119         end
120       else
121         % CASE 2
122         if (length(porth.Inport)==2)
123           indata= get_param(porth.Inport(1),'CompiledPortDimensions');
124           vector_size= indata(2);
125           % CASE 2 a
126           if ((vector_size) > 1)
127            f_block_h = [f_block_h;all_block_h(k)];
128             %disp(sprintf(' Input dimension of block %s inport 1 is %s',
                              block_name, vector_size));
129           else
130             % CASE 2 b
131             indata= get_param(porth.Inport(2),'CompiledPortDimensions');
132             vector_size= indata(2);
133             if ((vector_size) > 1)
134              f_block_h = [f_block_h;all_block_h(k)];
135              %   disp(sprintf(' Input dimension of block %s inport 1 is %s',
                              block_name, vector_size));
136             end
137           end
138         end
139       end
140     end
141   end
142 end

143 try
144   eval(sprintf('%s([],[],[],''term'')',model_name));
145 end

146 f_block_parent_h = mint_parent_h(f_block_h);

147 % Find all relevant subsystems
148 [dummy,subsys_i,dummy] = intersect(subsys_handle,f_block_parent_h);

149 % loop over subsystems
150 for j = subsys_i
151   % Check that the intersection is empty
152   blocks_i = find(subsys_handle(j) == f_block_parent_h);
153
154   if (isempty(blocks_i))
155     % set pass to the subsystem
156     subsys_name_s_c = mint_getfullname_c(subsys_handle(j));
157     mint_tdata_new_passmarkers(subsys_name_s_c, ...
158       scope_list_st, ...
159       1);
160   else
161     subsys_name_s_c = mint_getfullname_c(subsys_handle(j));
162     name_all_s = 'Limitations with regard to Operand Numbers for the Product Block';
163     info_s{1} = 'A product block with a fixed-point data type specified for the
output must comply with the following rules:';
164     info_s{2} = ' - the number of inports must not exceed two';
165     info_s{3} = ' - when driven by a vector the number of signals within this vector
must not exceed two.';
166     block_name_s_c = mint_get_param_c(f_block_h(blocks_i),'Name');
167     [sel_text_s_c, sel_handles_c] = ...
168       mint_make_selection2(name_all_s, ...
169       block_name_s_c, f_block_h(blocks_i));
170     mint_tdata_new_message(subsys_name_s_c, ...
171       mint_tdata_desc_get(scope_list_st, j), ...
172       sel_text_s_c, ...
173       sel_handles_c, ...
174       info_s);
175   end
176 end
```

Abbildung 7: Manuell erstelltes M-Skript mit derselben Funktionalität wie die Analyseregeln aus Abb. 4

Requirements Engineering in der Analysephase mit der Rational Suite

Michael Erskine
LFK-Lenkflugkörpersysteme GmbH, Unterschleißheim
michael.erskine@mbda-systems.de

Abstract: Dieser technische Beitrag beschreibt die Erfahrungen während der Analysephase aus einem Software-Projekt für eingebettete Echtzeit-Software in der Verteidigungsindustrie. Er gibt einen Überblick über die Analysemethode und ihre Umsetzung mit Hilfe der Werkzeuge Rose RT und RequisitePro.

1 Das Projekt

Bei dem Projekt handelt es sich um eingebettete Echtzeit-Software. Die Software wird auf einer Prozessorkarte eingesetzt, die in ein größeres System eingesteckt wird. Das Projekt wird in der Verteidigungsindustrie realisiert. Das Vorgehen basiert auf dem V-Modell 97. Objektorientierung und der Einsatz der UML sind Anforderungen des Auftraggebers. In dem Projekt arbeiten zwei Ingenieure, ein Mathematiker und ein Informatiker. Als Entwicklungszeit sind drei Jahre geplant. Die Größe der Software wird etwa 30.000 SLOC betragen.

Der Auftraggeber hat seine Anforderungen aus einem älteren, ähnlichen Projekt übernommen, modifiziert und ergänzt. Die Anforderungen sind insgesamt erfreulich vollständig und genau. Sie liegen in natürlichsprachlicher Form vor und folgen keiner expliziten Methode. Die Anforderungen umfassen mehrere Abstraktionsebenen: Die meisten beschreiben das System aus einer *Blackbox*-Sicht; manche fordern interne Zustände der Software und nehmen damit das Software-Design vorweg.

2 Die Analysemethode

Viele Autoren haben universelle Vorgehensweisen für die Objektorientierte Analyse definiert, die sich in den groben Schritten, in den einzelnen Methoden und in den Darstellungsarten zum Teil erheblich unterscheiden. Es fällt schwer, aus diesem Angebot das eine Vorgehen herauszufinden, das auf das eigene Projekt am besten zutrifft, das vom eingesetzten Werkzeug am besten unterstützt wird, das den Modellierungsaufwand gering hält und dennoch alle wichtigen Aspekte abdeckt.

Deshalb wurde die Analysemethode selbst entwickelt. Die Fachliteratur, zum Beispiel

[Oestereich 06], half dabei, die einzelnen Schritte zu identifizieren. Die nicht-trivialen Schritte wurden durch weiterführende Literatur verfeinert. Bei der Entwicklung der Methode wurde darauf geachtet, daß die Bausteine zusammenpassen und daß sie einen nachvollziehbaren Ablauf bilden. Um die Methode zu finden, wurde anhand von Beispielen aus dem realen Projekt eine kleine Menge Anforderungen prototypisch in einem separaten Modell umgesetzt.

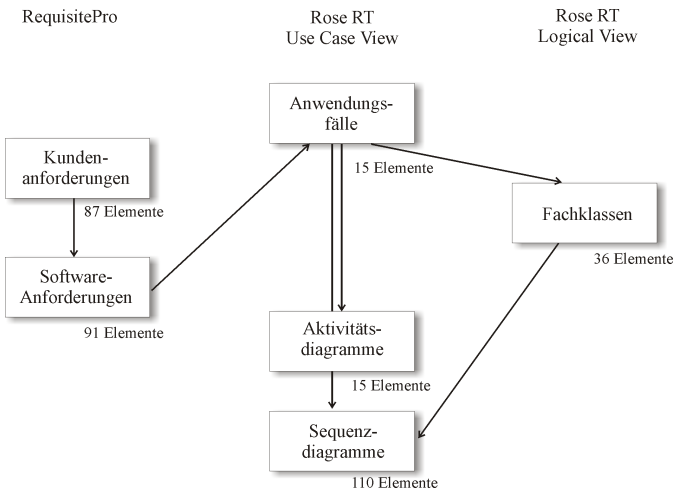


Abbildung 1: Die Schritte der Analysemethode

2.1 Verwalten der Kundenanforderungen

Der Auftraggeber lieferte seine Anforderungen in Form eines Word-Dokumentes. Das Originaldokument wurde in RequisitePro importiert. Es wurden 87 Anforderungen identifiziert, in RequisitePro markiert und mit wenigen zusätzlichen Attributen ausgestattet.

Die Arbeit mit RequisitePro fällt leicht. Das Werkzeug ist einfach, funktional und robust. In weniger als einer Stunde kann einem Mitarbeiter erklärt werden, wie Anforderungen erfaßt werden.

2.2 Ableiten von Softwareanforderungen

Um die unstrukturierten Anforderungen des Auftraggebers in eine strukturierte Form zu bringen und damit den ersten Schritt hin zu einer formalen Darstellung zu gehen, wurde die patternorientierte Methode der Sophisten benützt [Rupp 01]. Dabei wird jede Anforderung nach einer Schablone erstellt. Es existieren Schablonen für selbständige Systemaktivität,

Benutzerinteraktion und potentielle Fähigkeit des Systems.

Schablone für selbständige Systemaktivität:

Anforderung → [Wann?] [Unter welchen Bedingungen?] SOLL DAS SYSTEM <Objekt und Ergänzung> <Prozeßwort im Infinitiv>

Beispiel:

Wann?	Im Zustand „Standby“,
Unter welchen Bedingungen?	wenn das Signal „Start“ eintrifft,
	SOLL DAS SYSTEM
Objekt und Ergänzung	seinen Betriebszustand im Signal „Status“
Prozeßwort im Infinitiv	übertragen.

Die Spezifikation der 91 abgeleiteten Software-Anforderungen erfolgte in RequisitePro, jedoch mit einem anderen Anforderungstyp und ähnlichen, aber nicht identischen Attributen. Ein dritter Anforderungstyp enthielt das Abkürzungsverzeichnis, das Glossar und die Prozeßwortliste; diese drei Elemente werden durch ein Attribut unterschieden.

Prozeßworte sind Verben, die stellvertetend für einen Prozeß stehen, der eine vorgegebene Bedeutung hat. Dadurch werden Synonyme für die jeweils gleiche Aktivität vermieden. *Beispiel:* Statt „senden“, „verschicken“, „schicken“ wird „übertragen“ verwendet. In den Software-Anforderungen wurden 12 unterschiedliche Prozeßworte verwendet.

Mit der patternorientierten Methode wurden Lücken in den Anforderungen aufgedeckt. Die Methode führt zu eindeutigen Formulierungen, zu denen der Auftraggeber entweder *Ja* oder *Nein* sagen kann. Stellen, bei denen sich der Auftraggeber unsicher ist, werden durch *tbd* und *tbc* gekennzeichnet.

Den Kunden verwirrten die drei Schablonenarten. Die Schablonen für Benutzerinteraktion („Das System soll die Möglichkeit bieten“) und potentielle Fähigkeiten des Systems („Das System soll fähig sein“) stufte er als schwer nachweisbar ein. Deshalb wurde nahezu ausschließlich die Schablone für selbständige Systemaktivität verwendet („Das System soll“), das dem gewohnten Aussehen von Anforderungen am nächsten kommt. Die Qualität der Anforderungen litt darunter nicht. Der Nutzen der Schablonen liegt weniger in der Unterscheidbarkeit der drei Arten, sondern viel mehr in der schematischen Notation unter Verwendung definierter Begriffe.

Nur eine geringe, überschaubare Anzahl Anforderungen beziehen sich auf nichtfunktionale Aspekte. Deshalb wurde vorerst nicht zwischen funktionalen und nichtfunktionalen Anforderungen unterschieden. Sollte es später nötig erscheinen, ist es einfach, ein zusätzliches Attribut aufzunehmen.

2.3 Abstraktion zu Anwendungsfällen

Üblicherweise stehen Anwendungsfälle am Beginn der Analysephase. In diesem Projekt existierten bereits detaillierte Anforderungen, so daß es nicht notwendig war, mit dem Auftraggeber die grundsätzlichen Anforderungen herauszuarbeiten. Der hohe Detaillierungs-

grad führte jedoch dazu, daß man den eigentlichen Zweck der Software aus den Augen verlor: Man sah den Wald vor lauter Bäumen nicht mehr.

Deshalb wurden nachträglich 15 Anwendungsfälle in Rose RT erstellt und alle Software-Anforderungen in RequisitePro so gruppiert, daß die Anwendungsfälle sie abdecken.

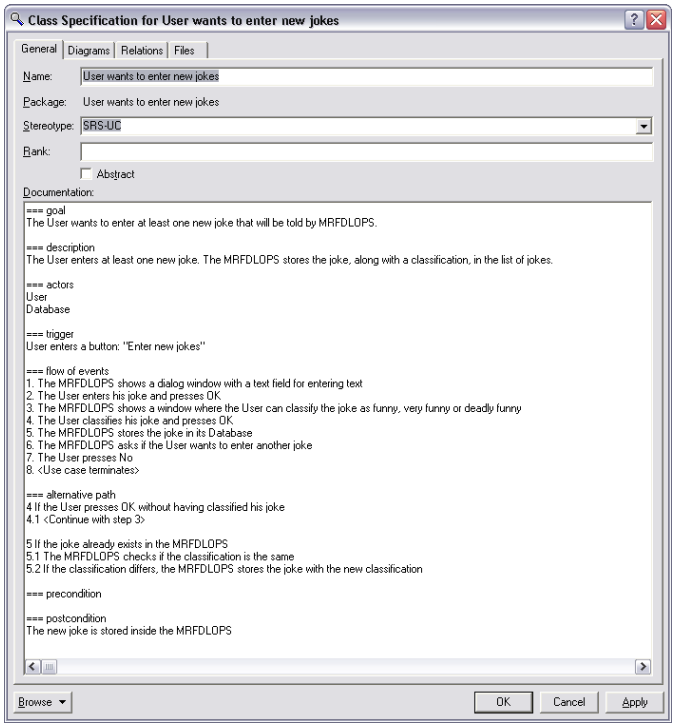


Abbildung 2: Ausgefüllte Schablone für Anwendungsfälle

Die Anwendungsfälle folgen einer vorgegebenen Schablone. Jeder Anwendungsfall enthält die Felder Ziel, Akteure, Auslöser, Vorbedingung, Hauptpfad, Alternativpfade und Nachbedingung. Die Erstellung orientierte sich an [Adolph et al. 03].

Die Anwendungsfälle dienten als Grundlage für ein technisches Review. Der Kunde konnte sich dabei davon überzeugen, daß die Entwickler die wesentlichen Aufgaben der Software korrekt verstanden haben. Bei der Durchsprache der einzelnen Schritte der Anwendungsfälle konnten einige Details korrigiert werden.

2.4 Finden von Fachklassen

Um für den nächsten Schritt die handelnden Objekte zu finden, wurden CRC-Karten erstellt, wie in [Wirfs-Brock et al. 03] beschrieben. Jede Karte enthält auf der Vorderseite

die Felder „Was stellt die Klasse dar?“ und „Warum ist die Klasse wichtig?“, auf der Rückseite die Felder „Verantwortlich für“ und „Arbeitet mit“.

Jeder Anwendungsfall diente als Ausgangspunkt für eine Simulation. Als CRC-Karten dienten selbsterstellte Karten im DIN A6-Format. Der bewußt knapp gehaltene Platz auf der Karte führte dazu, daß die einzelnen Klassenkandidaten nicht mit Aufgaben überladen wurden.

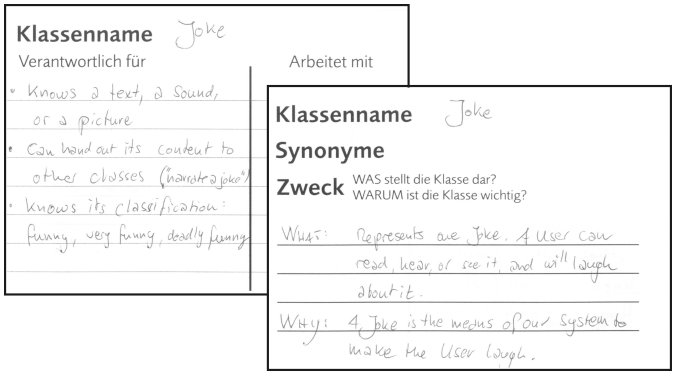


Abbildung 3: Selbsterstellte CRC-Karte

An einer Simualtion waren zwei bis vier Software-Entwickler beteiligt. Obwohl keiner der Beteiligten vorher die Methode kannte, wurde sie schnell verstanden und erfolgreich umgesetzt. Die Simulation führte zu einem besseren Verständnis des Problembereichs und der dynamischen Zusammenhänge. Nach der Simulation wurden die CRC-Karten an eine Pinnwand im Teamraum geheftet. Die Klassenkandidaten wurden als Fachklassen in Rose RT übertragen und durch 6 Stereotypen grob kategorisiert.

2.5 Verhaltensmodellierung

Für jeden Anwendungsfall wurde genau ein Aktivitätsdiagramm modelliert. Es zeigt den Hauptpfad sowie alle Nebenpfade, jedoch nicht alle Details aus dem Anwendungsfall.

Zusätzlich wurden Sequenzdiagramme modelliert, die exemplarisch das Zusammenspiel der Fachklassen aufzeigen. Die Sequenzen beschränken sich auf Details, insbesondere auf die Frage, wer ein Objekt erzeugt und löscht. Dabei erwiesen sich die sorgfältig ausgefüllten Felder auf den CRC-Karten als hilfreich. Bei jeder Operation kann man nachsehen, ob sie tatsächlich in die Verantwortung dieser Klasse fällt.

2.6 Dokumentation

Die Rational Suite bietet für die Erstellung von Dokumenten das Werkzeug SoDA an. Das V-Modell liefert Dokumentenvorlagen, bei denen bestimmte Inhalte an genau definierten Stellen stehen. Diese Vorlagen mit SoDA zu füllen, ist nicht effizient, da die SoDA-Skripte in diesem Umfeld unübersichtlich werden.

Deshalb wurden die V-Modell-Dokumente mit Hilfe von VBA erstellt. Sowohl Rose RT als auch RequisitePro bieten eine einfach programmierbare VBA-Schnittstelle. Der in Word eingebaute VBA-Interpreter erlaubt es, jeden noch so ausgefallenen Wunsch an die Dokumentation automatisch umzusetzen.

3 Erfolgsfaktoren

Neben der Vorgehensweise und den Werkzeugen wurde auf weitere Aspekte geachtet, so daß die Vorgehensweise erfolgreich umgesetzt werden konnte.

3.1 Konsistenz durch Regeln

Nachdem das Vorgehen gefunden war, wurde es in Form von festen Regeln fixiert. Die Regeln wurden in einer Modellierungsrichtlinie und einem *Requirements Management Plan* festgehalten. Jede Regel wird durch einen Regelchecker geprüft, der in VBA programmiert wurde. Die Regeln beschreiben den Aufbau der RequisitePro-Datenbank (Typen, Attribute, Views), den Aufbau des Rose RT-Modells (Pakete, Views) und den Einsatz der UML-Sprachelemente.

> *Beispiel für eine Anforderungsregel:* „Regel SWANF.TRACE: Eine SWANF-Anforderung muß auf mindestens eine AGANF getracet werden.“
>
> *Beispiel für eine Modellierungsregel:* „Regel UC.VIEW: Der Use Case View muß genau ein Paket mit dem Stereotyp «Actor Package» enthalten.“

Besonders viel Wert wurde auf Regeln gelegt, die die Konsistenz der Elemente sicherstellen. Die UML fördert die Darstellung des gleichen Sachverhalts aus verschiedenen Sichten mit unterschiedlichen Diagrammen. Die Werkzeuge erzwingen weder Vollständigkeit noch Konsistenz. Eigene Regelchecker können das nur in begrenztem Umfang kompensieren. Insbesondere das Prüfen von natürlichsprachlichen Elementen, wie zum Beispiel Anwendungsfälle, beschränkt sich auf formale Kriterien - in der Hoffnung, daß der Inhalt korrekt ist, wenn die Form stimmt.

3.2 Klassiker statt Experimente

Die Literatur ist voll von neuen, innovativen Ideen, wie man die UML einsetzen kann. Für den reinen Anwender ist es schwer, herauszufinden, was bewährt ist und was sich im Versuchsstadium befindet. Erst wenn ein Vorgehen in mehreren Büchern erwähnt wird, wie zum Beispiel die CRC-Karten, oder wenn man es auf UML-Klassiker wie [Booch 94] oder [Balzert 96] zurückführen kann, kommt es als Kandidat für die eigene Methode in Frage.

Die Herausforderung besteht darin, eine Methode über mehrere Schritte hinweg so zu finden, daß die einzelnen Schritte ineinander greifen. In diesem Projekt wurde auf bewährte Methoden für die einzelnen Schritte gesetzt.

3.3 Schablonen

Informatiker sind eine Minderheit unter den IT-Fachleuten [Dostal 02]. In der Verteidigungsindustrie arbeiten viele Ingenieure als Software-Entwickler. Ihre Herangehensweise ist durch das „ingenieursmäßige Vorgehen“ charakterisiert. Die Methoden berücksichtigen das, indem sie ein pragmatisches, schnell demonstrierbares Vorgehen erlauben und gleichzeitig auf soliden Methoden basieren. Wenn das Projekt erst einmal begonnen hat, bleibt kaum Zeit, um sich mit Theorien zu beschäftigen: Das Produkt steht im Vordergrund.

Als hilfreich haben sich Schablonen erwiesen, die vom Entwickler ausgefüllt oder angewendet werden. Schablonen kommen bei den Software-Anforderungen in Form der patternorientierten Methode zum Einsatz, bei den Anwendungsfällen in Form von Pflichtfeldern und bei den Fachklassen in Form von CRC-Karten mit Pflichtfeldern.

Ebenfalls hilfreich sind Beispiele, an denen sich der Entwickler orientieren kann. Ein typisches Beispiel sagt mehr als viele abstrakte Regeln. Der Entwickler kopiert das Beispiel und ändert es projektbezogen ab. Der Regelchecker prüft anschließend automatisch, ob das neue Element den Regeln entspricht.

4 Zusammenfassung

Eine konservative Branche wie die Verteidigungsindustrie ist von den klassischen Methoden wie der *Strukturierten Analyse* geprägt. Der Weg hin zur Objektorientierten Analyse erfolgte später als in anderen Branchen. Die Erwartung war, daß sich im Laufe der Zeit Standards etablieren: Man möchte sein Projekt grob charakterisieren nach Einsatzgebiet, Sprache, Größe, Dauer und in einer Tabelle nachsehen, welche UML-Artefakte nacheinander erstellt und welche Methoden eingesetzt werden müssen. Dann kann man sich voll und ganz auf die technischen Fragen konzentrieren.

Der Vorteil der Objektorienierten Analyse mit der UML ist gleichzeitig ihr Nachteil: sie ist universell. Der OOA-Anwender muß sich selbst genau überlegen, was er macht und wie er

es macht. Er muß die OOA auf sein Projekt zurechtschneiden. Um das zu können, muß er zum OOA-Spezialisten werden. Er muß wissen, was es für sein Vorgehen bedeutet, wenn er sich für ein Diagramm entscheidet und das andere wegläßt.

Bei der *Strukturierten Analyse* war es noch möglich, sich ein Werkzeug zu beschaffen und loszulegen. Das Werkzeug achtete schon darauf, daß man nicht zu viel falsch machte. Die Werkzeuge heute müssen, wenn sie die OOA mit der UML möglichst breit unterstützen wollen, ebenso universell wie Methode und Sprache sein. Wenn der Anwender will, daß das Werkzeug sein spezialisiertes Vorgehen optimal unterstützt, dann muß er selbst das Werkzeug programmieren.

Wer wie früher einfach „ingenieursmäßig“ vorgeht, wird in seiner Erwartung enttäuscht. Am Anfang des Projekts steht nun die intensive Auseinandersetzung mit Methoden, Sprachen und Werkzeugen. Dieser Aufwand wird heute noch unterschätzt. Lehrbücher und Theorien zur OOA gibt es viele. Woran es mangelt, sind detaillierte Erfahrungsberichte, von denen ausgehend man sein eigenes Vorgehen schneller und sicherer findet.

Alternativ könnten die Werkzeuge eine bessere Unterstützung verschiedener Vorgehen anbieten. Das Vorgehen sollte so auf Regeln abgebildet und im Werkzeug verankert werden, daß der Anwender per Knopfdruck erfährt, ob sein Diagramm syntaktisch korrekt ist. Die Rational Suite bietet das nicht; der Anwender muß seine Regeln selbst programmieren.

Aus meiner Sicht steht die Verteidigungsindustrie bei der Umsetzung der Objektorientierten Analyse mit UML noch am Anfang. Das Angebot ist vielfältig und ändert sich schnell. Das verunsichert den durchschnittlichen Anwender. In diesem Projekt ist der Aufwand für die Umsetzung der Methoden hoch; allerdings lohnt er sich, denn die Erfahrungen mit den Methoden sind gut.

Literatur

[Adolph et al. 03] Adolph, Steve; Bramble, Paul: *Patterns for Effective Use Cases*. 1. Auflage, Addison-Wesley, Boston, 2003.

[Balzert 96] Balzert, Helmut: *Lehrbuch der Software-Technik*. 1. Auflage, Heidelberg, Spektrum Akademischer Verlag, 1996.

[Booch 94] Booch, Grady: *Objektorientierte Analyse und Design*. 1. Auflage, Bonn, Addison-Wesley, 1994.

[Dostal 02] Dostal, Werner: *IT-Arbeistmarkt: Katastrophe oder Normalisierung?* Informatik Spektrum 25 (5), Springer, Heidelberg, 2002.

[Oestereich 06] Oestereich, Bernd: *Analyse und Design mit UML 2.1*. 8. Auflage, München, Oldenbourg, 2006.

[Rupp 01] Rupp, Chris: *Requirements-Engineering und -Management*. 1. Auflage, Carl Hanser Verlag, München, Wien, 2001.

[Wirfs-Brock et al. 03] Wirfs-Brock, Rebecca; McKean, Alan: *Object Design. Roles, Responsibilities, and Collaborations*. 1. Auflage, Addison-Wesley, Boston, 2003.

Simulation-Driven Creation, Validation and Evolution of Behavioral Requirements Models

Martin Glinz Christian Seybold[1] Silvio Meier

Institut für Informatik
Universität Zürich
Binzmühlestrasse 14
CH-8050 Zurich, Switzerland
{glinz | smeier}@ifi.unizh.ch
cseybold@gmx.ch

Abstract: Requirements models for large systems cannot be developed in a single step; they evolve in a sequence of iterations. We have developed a simulation-driven process that supports iterative, evolutionary modeling of behavioral requirements. We start with modeling type scenarios (i.e. use cases) and simulate these interactively. The simulation runs yield exemplary system behavior, which is documented in message sequence charts (MSCs). The modeler can then generalize this recorded partial behavior into statecharts. The resulting model is simulated again for validating that the modeled behavior matches the previously recorded behavior. The validated model is then used in the next incremental step for eliciting new, yet unspecified behavior by simulating new scenarios.

1 Introduction

Requirements models under development are typically incomplete and not completely formalized. In an iterative development process, modeling proceeds by progressively making models more complete and more formal. This process of model evolution stops when the model has reached the desired degree of formality and completeness (depending on risk, time and budget). In order to support such a process, three requirements must be met: we need (i) a modeling language that provides features for modeling intentional partial incompleteness and a variable degree of formality, (ii) a technique for early and frequent model validation, and (iii) guidance for systematically eliciting and evolving models towards more formality and more completeness.

The ADORA language [GBJ02] that has been developed in our research group satisfies the first requirement.

For early model validation, simulation would be a quite appropriate technique both for modelers and stakeholders: they can play with the model's dynamics by entering stimuli

[1] Christian Seybold is now with Zühlke Engineering (Switzerland)

and receiving system reactions. However, simulation, as well as other automated techniques (e.g. model checking) cannot be applied as long as we do not have complete and formal models. Therefore, we have developed an interactive simulation technique that allows to simulate incomplete and semi-formal models by inquiring missing information interactively from the expert who runs the simulation. The information provided by the expert is recorded so that regression simulation becomes possible [SMG05]. This technique satisfies the second requirement.

Our simulation technique also enables an iterative, outside-in development process for behavioral requirements models that starts with some external behavior specified by type-level user-system interaction scenarios (aka use cases) and progressively elicits and defines system behavior with simulations that are driven by playing through the scenarios [SMG04], [SMG06], which addresses the third requirement.

In this paper we summarize our approach, which is documented in [Se06], [SMG04], [SMG05] and [SMG06], and give an example.

2 Our approach

2.1 Simulation[2] of semi-formal models

We use ADORA [GBJ02] as a modeling language. Models in ADORA are composed of hierarchically structured abstract objects. Each object represents a state and may be further decomposed by other objects and by embedded statecharts. All objects and states together form one joint, hierarchical statechart. However, our approach is not restricted to ADORA. Alternatively one could use a sufficiently formalized UML profile. The advantage of ADORA is that it already has the features we need, whereas with UML, one first had to define and formalize a suitable subset.

The simulation engine of the ADORA tool simulates models specified in ADORA regardless of their degree of formality and completeness. In all situations where the simulator can't interpret the model because information is missing or the given information is not formal enough, the simulation engine lets the modeler interactively specify the desired behavior. That means, the simulation engine executes the specified system behavior, i.e. on occurring events it performs transitions between states and executes specified actions. As soon as an event appears that cannot be handled, the simulation is interrupted to allow the user to handle this event interactively: whether it shall be received at all, by which object and which actions shall be performed on this event. Afterwards, the simulation continues as usual. Fig. 1 shows a screenshot of the ADORA tool during an interactive simulation session. The simulation engine is currently executing the Input Value scenario in a partial model of a calculator. As the model does not specify whether the Add or the Subtract scenario shall be executed next, the simulation engine lets the mod-

[2] When we talk about simulation, we mean an event-driven, discrete simulation. We do not consider real-time or continuous simulations.

eler interactively select an alternative. More details on our interactive simulation engine can be found in [SMG05] and [Se06].

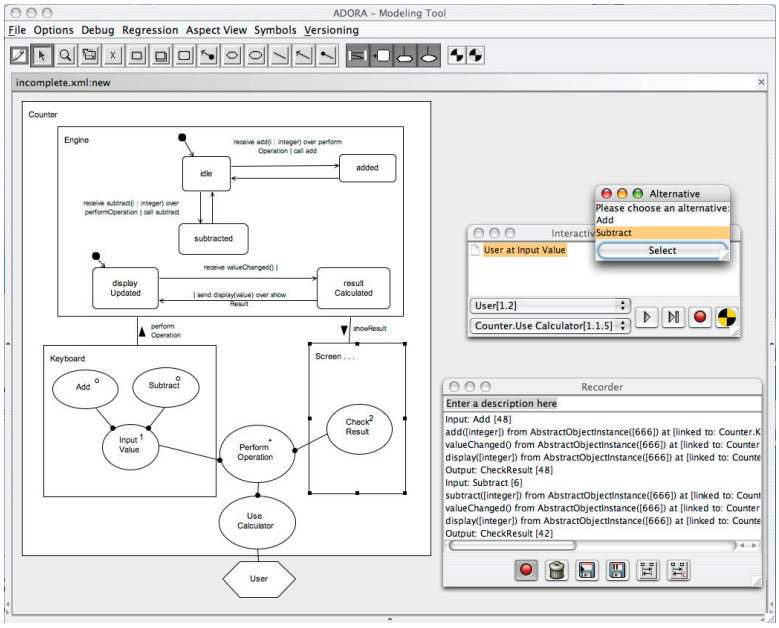


Figure 1: A screenshot of the ADORA tool during an interactive simulation session

The events driving the simulation are generated by playing through the scenarios of the model. Scenarios in ADORA describe the communication protocol between the modeled system and the actors in the context of the system, i.e. the behavior as it is seen by the external actors. Note that scenarios in ADORA are *type scenarios*, i.e. use cases in UML terminology. In ADORA, scenarios are modeled with so-called scenariocharts, a notation that is derived from Jackson's JSP diagrams [Ja75] and is capable of modeling scenario decomposition in terms of sequence, alternative, iteration and parallelism of sub-scenarios [Gl95], [GBJ02]; see Fig. 2.

For each modeled actor, the user simulating the model may create an instance, thereby launching the traversal of the connected scenariochart. The simulation stops at leaf nodes allowing the user simulating the model to enter stimuli or receive system reactions. The graphical interface to enter stimuli is automatically built from the transform expressions specified at the leaf nodes of the scenariocharts.

As every small modification of a requirements model can unintentionally destroy required properties that held in the model prior to the modification, a model should be revalidated after each incremental step. In our approach, revalidation is done by *regression simulation*, i.e. automatically re-executing previously recorded simulation runs. While the principal idea is fairly obvious, some major problems have to be solved to

make regression simulation work in our context. In particular, regression simulations have to run automatically regardless of the amount of interactivity that was required to generate the original simulation runs. We achieve this by letting the model drive the regression simulation, but automatically resort to the interactively recorded behavior in those cases where the simulation engine does not get enough information from the model during a regression simulation run.

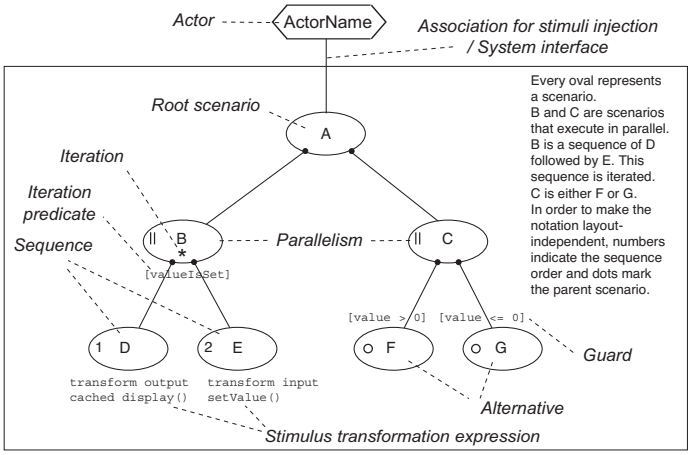


Figure 2: A scenariochart and its interpretation

For every completed simulation run, the ADORA tool can automatically generate a Message Sequence Chart [MSC96]. These MSCs can be used for analyzing simulations (in particular, when a regression simulation fails [SMG05]) and for driving the incremental modeling process (see next sub-section).

2.2 Iterative development of behavior models

Models of system behavior can be created in two ways: (a) the modeler begins with chunks of exemplary behavior which are then synthesized and generalized, (b) she or he creates a generalized model of behavior (i.e. a type level model that covers all potential cases) from the beginning and validates it by examining how the model behaves when playing through specific, exemplary cases.

Exemplary behavior is cognitively easier to develop and also better suited for discussing models with stakeholders for validation purposes. So it is a good starting point for system modeling. However, as exemplary behavior just gives an outline of certain situations, we eventually need type-level behavior models. On the other hand, creating type-level models from scratch is difficult and requires considerable effort and expertise.

An obvious solution would be generating type-level models automatically from a collection of exemplary, instance-level models (e.g. synthesizing statecharts from MSCs

[WS00], [KGSB99]) However, these approaches have drawbacks. Firstly, the generated statecharts are typically hard to read for humans and, hence, difficult to validate. Secondly, when we interpret MSCs existentially[3], there is typically more than one statechart that describes the behavior specified by a given set of MSCs. So there is no unambiguous mapping from MSCs to statecharts (unless one introduces a special syntax and semantics for MSCs, as done in [KGSB99].

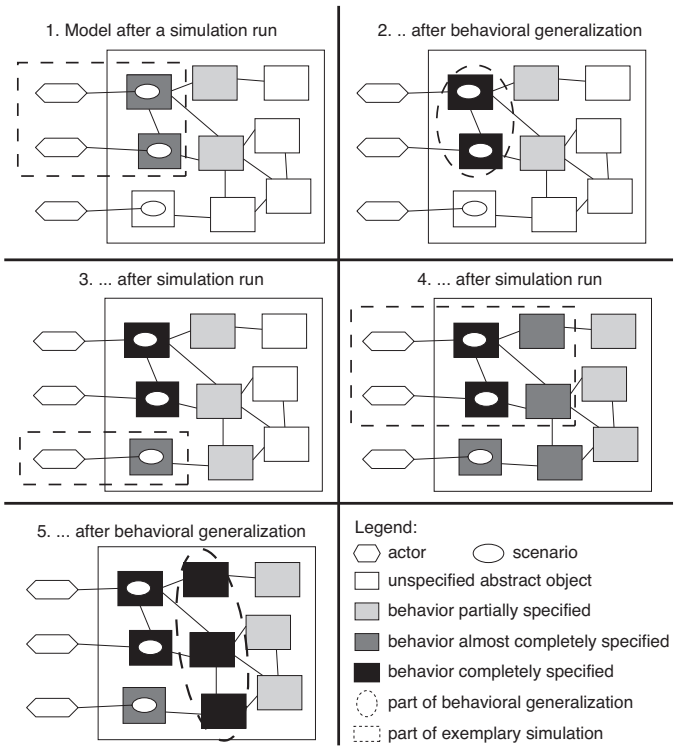


Figure 3: Incrementally building a behavior model

Therefore we decided to pursue a manual, tool-guided approach which combines exploration of exemplary behavior with systematic (but manual) construction of statecharts. Starting from scenarios describing external behavior as seen by actors, we derive exemplary behavior by running simulations on a void or incomplete behavior model and define the expected behavior interactively. For every such simulation run, the ADORA tool generates a message sequence chart. These MSCs are used to guide and inform a manual process of constructing statecharts that exhibit the expected behavior when executed with the given scenarios. Regression simulation is used to validate the created statecharts systematically.

[3] Existentially means that the specified behavior must be possible, but other behavior may also occur. This is the usual MSC semantics.

Having specified parts of the desired behavior with statecharts, subsequent simulation runs become easier: the modeled behavior is integrated into the simulation, thus making it more precise and requiring less interactivity during simulation runs.

When used this way, the two modeling modes, viz. recording exemplary system behavior by playing through scenarios and modeling system behavior with statecharts, i.e. on a type level, stimulate each other and drive the model development towards more complete and more formal specifications. Fig. 3 illustrates how an alternating application of simulation runs and behavioral generalization/synthesis contributes to the evolution of a system model.

3 Example

We illustrate our approach with an example (taken from [SMG06] and [Se06]). As an example application, we refer to the specification of a control system for car doors given in [HP02]. A first version of the model is given in Fig. 4. Some scenarios have been modeled already whereas the system is yet unspecified except that it has been partitioned into the components Seat Adjustment, Door Locking and User Management.

In a first step, we decide to focus on the seat positioning via the user management switches. There are four switches inside the car to recall four different seat positions (for different drivers). A seat may be adjusted in five different ways: seat height in the front and back, angle of the back, seat distance to the wheel, and tightness of the casing. While being seated, seat adjustments must be done in a comfortable way, i.e. the relaxing adjustments must happen before constrictions take place (in order not to trap the driver), and not more than two movements may be done at the same time. However, when unlocking the car with a radio transmitter (there are two different ones for two drivers), the seat position shall be adjusted as fast as possible to be ready before the driver enters the car. The switches inside the car are connected to the door control system via interface S1, the radio transmitter uses the CAN-bus for communication.

In our example, we start with a simulation of a typical sequence of interactions how the seat adjustment could take place, executing the CAN Bus Control scenario first (i.e. a user unlocks the car) and then the User Management Scenario (i.e. a user presses seat adjustment switches). At that time, there is no behavior specified in the system. All objects taking part in the simulation (CAN, S1, User Management and Seat Adjustment) are being played by the modeler. This means that for each incoming event, the modeler specifies the corresponding actions that should take place. The result of this simulation run is shown in Fig. 5. As we are currently concentrating on the user management, we are not interested in the actual seat positioning. That is why we do not continue to handle the messages in the Seat Adjustment object (marked with a cross in Fig. 5). This may be the focus of further simulations. Therewith, we have outlined some exemplary behavior for the involved objects. We could either proceed by recording more simulation runs to enrich the system with exemplary behavior or we continue with the generalization of the existing behavior.

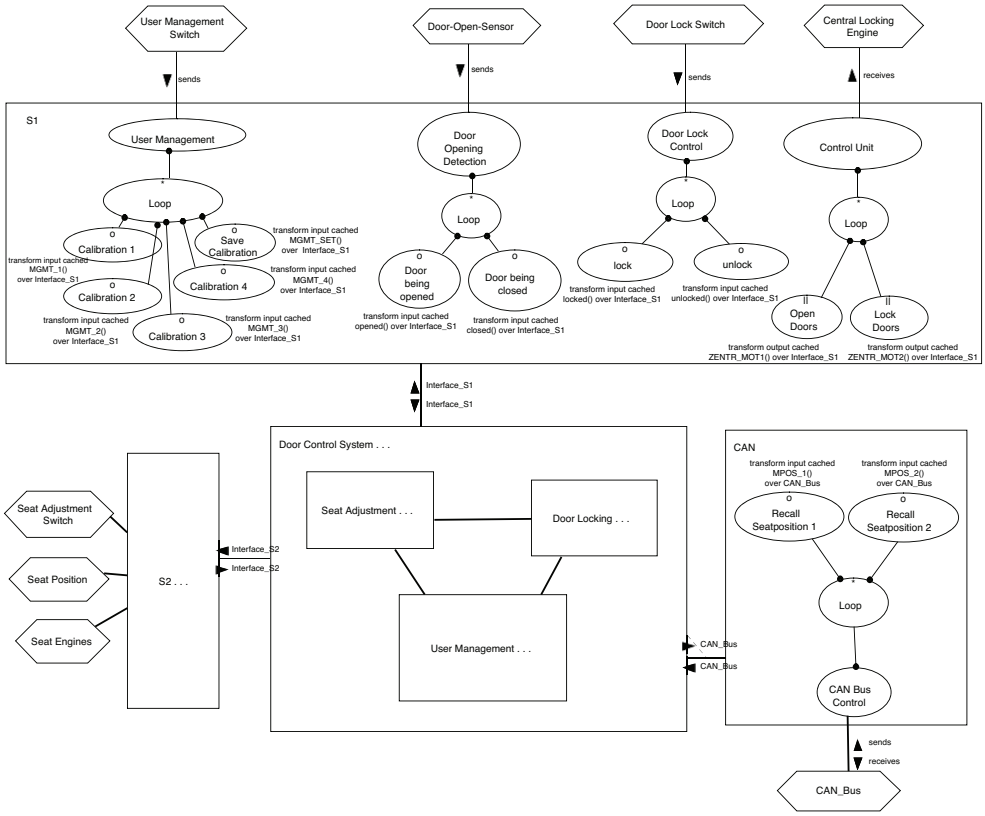


Figure 4: Initial model of a car door locking system

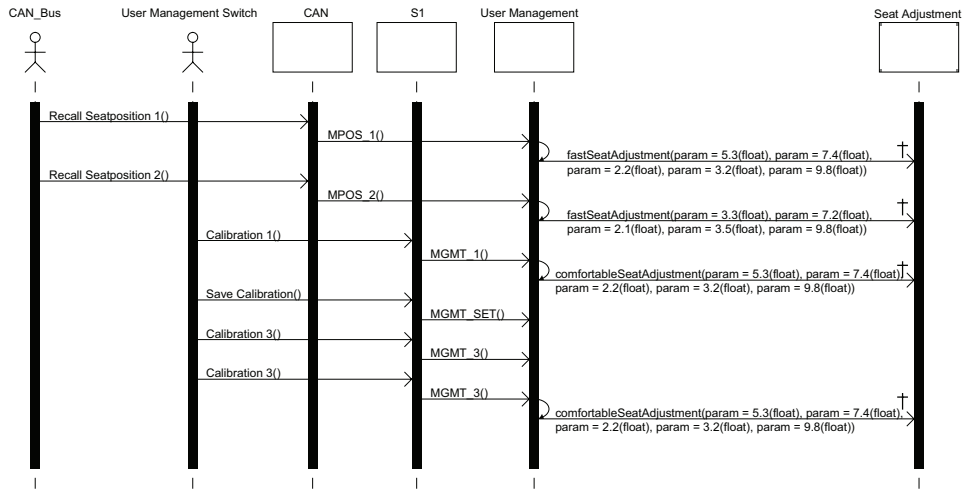


Figure 5: Trace of an exemplary play-through of the user management seat adjustment scenarios (automatically generated by the ADORA tool)

We have done the latter for the User Management object in Fig. 6. We selected the User Management lifeline from the MSC in Fig. 5 and created states and state transitions such that all incoming and outgoing messages are handled. This example also illustrates why we prefer a manual process: we cannot infer automatically whether the order of messages in the MSC matters or not. For modeling an adequate statechart, we hence need additional knowledge (domain knowledge or information elicited from stakeholders). In our case we found that the order of messages does not matter, leading to the statechart given in Fig. 6. Up to now, this statechart can exactly handle the recorded sequence chart, nothing else. However, for all following simulations, the specified behavior does not need to be played by the user any more. It will be taken from the statechart instead. Only new behavior must be played by the user.

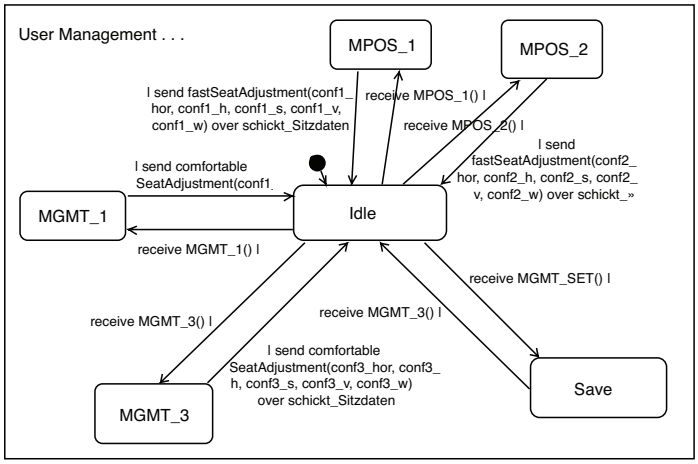


Figure 6: Generalized behavior of sequence chart in Fig. 5 in object User Management

This allows the modeler to focus on the specification of new behavior, for example enrich the behavior of the User Management Object. Simulations and behavioral generalizations can alternately take place, driving the model evolution until the model eventually exhibits the desired behavior.

4 Related Work

To the best of our knowledge, there is only one approach which is closely related to ours: Harel et al. developed the Play-Engine [HM03] that allows to play and test behavior of an incomplete component interactively via a prototypically built user interface. Harel focuses on the interface being developed whereas we are focusing the model being developed. Our main focus lies on adequate support of the requirements engineer in the modeling process. The model can be executed in any state of completeness to drive the further development of the model.

The idea of building models from exemplary behavior follows a rather old idea which was introduced by Shapiro for synthesizing logic programs from examples [Sh83].

Simulation as a means of validating a model is not a replacement for model checking approaches (e.g. [Ho97]). Simulation is used earlier in the process to validate the model and drive the further development of the model when it is not yet formal and complete enough to allow model checking.

5 Conclusions

In this paper, we have summarized and demonstrated our approach to simulation-driven modeling, validation and evolution of requirements models, using the ADORA language and tool.

The modeling and simulation capabilities described in this paper have been implemented in the ADORA prototype tool. This is a standalone system implemented in Java. Currently, the ADORA tool is being re-implemented as an Eclipse plug-in, which allows us to re-use a lot of existing model editor features. The re-implementation of the basic tool functionality has been completed. However, the re-implementation of the simulation engine has yet to be done. Our future work will concentrate on the incorporation of aspect modeling into ADORA and on further enhancing the modeling capabilities of both the language and the tool.

References

[Gl95] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In W. Schäfer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*. Lecture Notes in Computer Science Vol. 989, Springer, 1995. 254-271.

[GBJ02] M. Glinz, S. Berner, and S. Joos. Object-Oriented Modeling with ADORA. *Information Systems* 27(6):425-444, 2002.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer, New York, 2003.

[Ho97] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering* 23(5):279-295, 1997.

[HP02] F. Houdek and B. Paech. *Das Türsteuergerät – eine Beispielspezifikation* [The Car Door Control System – An Example Specification (in German)]. Technical report, Fraunhofer Institut für Experimentelles Software Engineering, Kaiserslautern, 2002.

[Ja75] M. Jackson. *Principles of Program Design*. Academic Press, New York, 1975.

[KGSB99] I. Krüger, R. Grosu, P. Scholz, M. Broy. From MSCs to Statecharts. Proceedings of the IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems, Schloss Eringerfeld, Germany, 1999. 61 - 71.

[MSC96] Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.

[Se06] C. Seybold. *Simulation teilformaler Anforderungsmodelle* [Simulation of semi-formal requirements models (in German)] PhD Thesis, University of Zurich, 2006.

[Sh83] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, 1983.

[SMG04] C. Seybold, S. Meier, and M. Glinz. Evolution of Requirements Models by Simulation. *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE'04)*, Kyoto, Japan, 2004. 43-48.

[SMG05] C. Seybold, S. Meier, and M. Glinz. Simulation-Based Validation and Defect Localization for Evolving, Semi-Formal Requirements Models. *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, Taipei, Taiwan, 2005. 408-417.

[SMG06] C. Seybold, S. Meier, M. Glinz. Scenario-Driven Modeling and Validation of Requirements Models. *Proceedings of the 5th ICSE International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM 2006)*. Shanghai, China, May 2006. 83-89.

[WS00] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. *Proceedings of the 24th International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000. 314-323.

Generierung von UML-Modellen aus formalisierten Anwendungsfallbeschreibungen

Mario Friske, Bernd-Holger Schlingloff
Fraunhofer FIRST
Kekuléstraße 7
D-12489 Berlin
{mario.friske|holger.schlingloff}@first.fraunhofer.de

Abstract: In diesem Kurzbeitrag zeigen wir, wie textuelle Anwendungsfallbeschreibungen werkzeuggestützt in UML-Aktivitätsdiagramme überführt werden können. Mit Hilfe eines von uns entwickelten Werkzeuges formalisieren wir zunächst die Beschreibung der Interaktionen, um anschließend aus dem so erstellten formalisierten Anwendungsfallmodell mittels automatisierter Transformation ein UML-Modell zu generieren. Die notwendige Transformation definieren wir in abstrakter Form und setzen diese mit verschiedenen Transformationswerkzeugen um.

1 Motivation

Anwendungsfälle (engl. Use Cases) sind ein weit verbreitetes Mittel zur Strukturierung und Spezifikation funktionaler Anforderungen an Softwaresysteme. Ein Anwendungsfall definiert eine Menge von Interaktionen zwischen Akteuren und dem System, die dazu dient, ein bestimmtes fachliches Ziel zu erreichen. Anwendungsfälle können als textuelle Beschreibungen oder in Form von Diagrammen, z. B. als Sequenz- oder Aktivitätsdiagramme, oder als Kombination von Text und Diagrammen spezifiziert werden.

Oftmals werden Anwendungsfälle als erster Schritt einer Systementwicklung in der Analysephase erstellt, um einen Überblick über die geforderte Systemfunktionalität zu gewinnen. Gängige Werkzeuge zur Anforderungsanalyse (z. B. Telelogic DOORS) unterstützen die Verwaltung und Verlinkung von Anwendungsfällen, erlauben es jedoch nicht, diese direkt für die Systementwicklung oder den Systemtest weiter zu verarbeiten.

Die interaktive Überführung von informellen Anwendungsfallbeschreibungen in eine formale Zwischenrepräsentation wurde von uns in einer früheren Publikation [FS05] diskutiert. In der modellbasierten Entwicklung werden häufig grafische Darstellungsformen verwendet, insbesondere die Unified Modeling Language (UML). Die grafische Darstellung eines Ablaufs als Diagramm bietet oftmals den Vorteil des leichteren Überblicks. Eine einfache Möglichkeit der Überführung von formalisierten textuellen Beschreibungen in eine grafische UML-Repräsentation wäre daher wünschenswert.

2 Der Use Case Validator

Mit dem *Use Case Validator* (UCV) wird bei Fraunhofer FIRST ein Werkzeug zum anwendungsfallbasierten Systemtest entwickelt. Ziel ist dabei unter anderem der Blackbox-Test eingebetteter Systeme. Der UCV basiert auf der *Eclipse*-Plattform [Ecl] und verfolgt einen modellbasierten Ansatz, in welchem Datenstrukturen durch Metamodelle definiert sind und mittels Modelltransformationen weiterverabeitet werden. Als Grundlage verwenden wir das *Eclipse Modeling Framework* [BEG+03] und darauf basierende Transformationsmaschinen, von denen wir zu Experimentierzwecken mehrere in den UCV eingebunden haben [FH06].

Der UCV ermöglichst es dem Nutzer, Anwendungsfallbeschreibungen unter Verwendung von Entwurfsinformation interaktiv zu formalisieren. Zur Veranschaulichung greifen wir das Beispiel aus [FS05] wieder auf, den in Abbildung 1 gezeigten Anwendungsfall „Record a Message“. Dieses ursprünglich aus [PL99] stammende Beispiel ist Teil der funktionalen Spezifikation eines digitalen Sound-Recorders und dient dort zur Veranschaulichung einer UML-basierten Entwicklungmethodik.

```
1. The user selects a message slot from the message
   directory.
2. The user presses the 'record' button.
3. If the message slot already stores a message, it is
   deleted.
4. The system starts recording the sound from the
   microphone until the user presses the 'stop' button,
   or the memory is exhausted.
```

Abbildung 1: Der Anwendungsfall „Record a Message“ in Textform

Zur Formalisierung mit dem UCV werden zunächst Systemfunktionen und -reaktionen bestimmt. Danach wird der Kontrollfluss ermittelt und anschließend die Systemfunktionen den Ablaufschritten zugeordnet. Dabei werden schrittweise die Mehrdeutigkeiten der textuellen Repräsentation interaktiv aufgelöst. Bei der Formalisierung des Kontrollflusses wird intern ein Kontrollflussmodell aufgebaut, welches für den Anwender des UCV nicht unmittelbar sichtbar ist. In der in Abbildung 2 gezeigten Benutzeroberfläche wird der Kontrollfluss nur durch Umklammerung und Einrückungen von Schritten repräsentiert, was in Abbildung 3 nochmal im Detail dargestellt wird. Die einzelnen Ablaufschritte werden formalisiert, indem ihnen durch den Anwender aufrufbare Systemfunktionen und mögliche Systemreaktionen zugeordnet werden, siehe auch [FS05]. Systemfunktionen- und reaktionen können optional auch Parameter enthalten, an welche Variablen gebunden werden. Im Beispiel wird auf diese Art und Weise durch die Variable `Slot` der Datenaustausch zwischen den Schritten realisiert. Das Ergebnis der Bearbeitung mit dem UCV ist ein formalisiertes Anwendungsfallmodell, welches durch das in [FP05] beschriebene Metamodell definiert ist und den Kontrollfluss und die einzelnen Ablaufschritte umfasst.

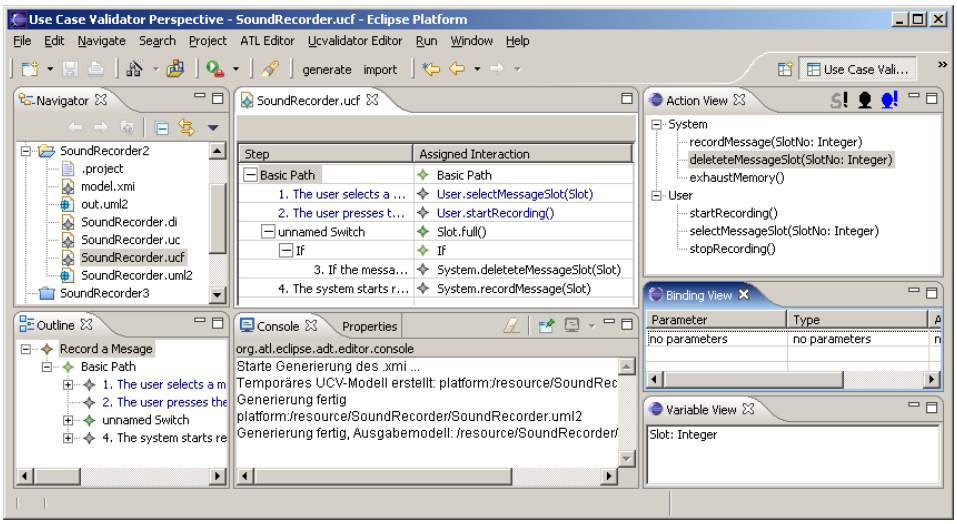


Abbildung 2: Die Benutzeroberfläche des Use Case Validator

```
User.selectMessageSlot(Slot)
User.startRecording()
if(Slot.full())
  System.deleteMessageSlot(Slot)
Exception(System.exhaustMemory()||User.stopRecording())
  System.recordMessage(Slot)
```

Abbildung 3: Die Formalisierung des Anwendungsfalls „Record a Message“

3 Generierung von UML-Modellen

Aus diesen formalisierten Anwendungsfällen wollen wir nun UML-Aktivitätsdiagramme mittels Modelltransformation generieren. Dieses Problem ist vergleichbar zur Generierung von Kontrollflussgraphen aus Programmen. Mit dem vorgestellten Ansatz können beliebige, durch ein Metamodell definierte, visuelle Repräsentationen generiert werden. UML-Aktivitätsdiagramme sind eine natürliche und weit verbreitete grafische Darstellungsform für Anwendungsfälle.

Die Transformation spezifizieren wir zunächst als abstrakte Transformationsbeschreibung, um eine einfache Abbildung auf verschiedene Modelltransformationssprachen zu ermöglichen. Diese Transformationsbeschreibung haben wir in Anlehnung an [ATL05] erstellt. Der Pseudocode für die Transformation ist in Abbildung 4 dargestellt. Er beschreibt die Abbildung von Elementen im Ausgangsmodell, dem Metamodell des UCV [FP05], auf Elemente im Zielmodell, dem Metamodell des Eclipse-UML2-Projekts [Hus04]. Eine einzelne Transformationsregeln ist dabei folgendermaßen zu lesen: Für jedes links beschriebene Element aus dem Ausgangsmodell sind alle rechts von den zugehörigen Pfeilen (⇒)

UCF2UML-AD: Formalisierte Anwendungsfallbeschreibung in UML2-Aktivitätsdiagramm

Kurze Beschreibung: Die Transformation UCF2UML-AD generiert UML2-Aktivitäten für alle formalisierten Anwendungsfälle im Ausgangsmodell. Diese Aktivitäten modellieren den Kontrollfluss der Anwendungsfälle.

Ausgangs-Metamodell: Metamodell des Use Case Validator: http://ucvalidator.ecore

Ziel-Metamodell: UML 2.0: http://www.eclipse.org/uml2/1.0.0/UML
Webseite des Eclipse-UML2-Projekts: http://www.eclipse.org/uml2

Transformationsregeln:

• ***UCFModel***	⇒	***Model***
• ***UCFormalization*** mit enthaltenem ***Block***	⇒	namensgleiche ***Activity***, enthalten in obigem ***Model***; alle anderen zu erstellenden Elemente sind in dieser ***Activity*** enthalten, soweit nicht anders angegeben
	⇒	***ActivityPartition*** für die Systemreaktionen
	⇒	***InitialNode*** und ***ActivityFinalNode***
	⇒	***ControlFlow*** vom ***InitialNode*** zu aus dem Block erstellten Knoten, von dort zum ***ActivityFinalNode***
• ***Step*** mit enthaltener ***Interaction***	⇒	***CallOperationAction*** namensgleich mit ***Interaction***
• ***Sequence*** mit enthaltenen Blöcken	⇒	***ControlFlow*** zwischen aus den Blöcken erstellten Knoten
• ***Switch*** mit enthaltenen Blöcken	⇒	***DecisionNode***
	⇒	***MergeNode***
	⇒	***ControlFlow*** zwischen ***DecisionNode***, aus den Blöcken erstellten Knoten und ***MergeNode***
• ***Loop*** mit enthaltenem ***Block***	⇒	***MergeNode***
	⇒	***DecisionNode***
	⇒	***ControlFlow*** von ***MergeNode*** zu aus dem ***Block*** erstellten Knoten, von dort zum ***DecisionNode***, von dort zum ***MergeNode***
• ***Actor***	⇒	***ActivityPartition***
• ***Variable***	⇒	namensgleicher ***CentralBufferNode***
• ***Parameter***, in ***Interaction*** und damit ***Step*** enthalten	⇒	namensgleicher ***InputPin*** bzw. ***OutputPin*** enthalten in erstellter ***CallOperationAction***
• ***Binding*** mit Referenzen auf ***Variable*** und ***Parameter***	⇒	***ObjectFlow*** zwischen aus den Referenzen erstelltem ***CentralBufferNode*** und ***Pin***

Abbildung 4: Abstrakte Transformationsspezifikation mittels Pseudocode

Elemente im Zielmodell zu erstellen. Dabei sind ggf. die in textueller Form beschriebenen Anweisungen zu berücksichtigen.

Die so spezifizierte Transformation erstellt für jeden formalisierten Anwendungsfall (***UC-Formalization***) eine Aktivität (***Activity***). In der Aktivität wird für jeden Ablaufschritt (***Step***) eine Aktion (*CallOperationAction*) und für jede Variable (***Variable***) ein Pufferknoten (***CentralBufferNode***) angelegt. Objektfluss und Kontrollfluss werden getrennt überführt. Der Kontrollfluss wird durch Kontrollflusskanten (***ControlFlow***) und Kontrollknoten (***DecisionNode, MergeNode***) realisiert. Der Objektfluss wird so umgesetzt, dass Pufferknoten durch Objektflusskanten (***ObjectFlow***) und Pins (***InputPin, OutputPin***) mit den Aktionen verbunden werden.

Dieser Pseudocode kann in ausführbaren Code für Transformationsmaschinen umgesetzt werden. Im UCV ist dies mit vier verschiedenen EMF-basierten Werkzeugen realisiert worden, welche mittels Wrapper-Plugin eingebunden wurden [Hil06]. Ein Erfahrungsbericht und Vergleich der verschiedenen Transformationswerkzeuge ist in [FH06] zu finden. Im vorliegenden Papier konzentrieren wir uns auf die Anwendung der Transformationssprache ATL [JK05], da sie frei verfügbar ist, zur Zeit stark weiterentwickelt wird und somit die Möglichkeit für weitere Experimente bietet.

Als Beispiel geben wir in Abbildung 5 einen Ausschnitt des ATL-Codes. Die gezeigte Transformationsregel überführt eine bedingte Verzweigung ***Switch*** in entsprechende UML-Konstrukte: ein Entscheidungsknoten ***DecisionNode*** und ein Ausgangknoten ***MergeNode*** werden angelegt und durch Kontrollflusskanten ***ControlFlow*** verbunden.

Das Ergebnis der Transformation einer formalisierten Anwendungsfallbeschreibung ist ein UML-2.0-Modell als Instanz des Metamodells des Eclipse-UML2-Projekts [Hus04]. Es liegt als Ecore vor und kann daher ohne weiteres als XMI-Datei serialisiert werden. XMI ist ein weit verbreitetes Format zum Speichern von Modellen [Obj03]. Das generierte Modell enthält die dem Anwendungsfall zugeordnete Aktivität, d.h. ihren Namen, die zugehörigen Aktivitätspartitionen, Aktionen und den Kontrollfluss. Abbildung 6 zeigt einen Ausschnitt der generierten XMI-Datei, in welchem die eine bedingte Verzweigung bildenden UML-Elemente dargestellt sind.

Dieses UML-Modell kann dann zur Visualisierung in ein handelsübliches UML-Modellierungswerkzeug geladen werden. Abbildung 7 zeigt eine solche Visualisierung des Transformationsergebnisses für das obige Beispiel, welches in diesem Fall mit Borland Together 2006 [Bor] erstellt wurde. Together bietet ein Autolayout für automatisch generierte Diagramme an, welches allerdings nur für sehr einfache Diagramme befriedigende Ergebnisse liefert. Deshalb wurde das Layout per Hand optimiert, um die Lesbarkeit zu verbessern. Die aktuell verfügbaren UML-Werkzeuge implementieren die in der UML2.0 definierten Sprachkonstrukte noch nicht in vollem Umfang, so wurde beispielsweise das Konstrukt *ExceptionHandler* in keinem der von uns betrachteten Werkzeugen unterstützt.

```
rule Switch2Node1 {
    from switch : ucv!Switch(switch.ElseBlock
                             ->oclIsUndefined())
    to
     -- Der Entscheidungsknoten
     dec  : uml!DecisionNode (
        name <- switch.Condition.description
     ),
     -- Der Mergeknoten üfhrt die Pfade wieder zusammen,
     -- ist der Ausgangsknoten einer bedingten Anweisung
     out : uml!MergeNode(
        name <- 'end'+switch.name
     ),
     e1 : uml!ControlFlow(
        source <- dec,
        target <- switch.IfBlock.getValidBlocks()->first()
     ),
     e2 : uml!ControlFlow(
        source <- dec,
        target <- out
     ),
     e3 : uml!ControlFlow(
        source <-  thisModule.resolveTemp(
                        switch.IfBlock.getValidBlocks()
                        ->last(),'out'
                    ),
        target <- out
     )
}
```

Abbildung 5: Eine Transformationsregel aus der ausführbaren ATL-Transformation

4 Diskussion und Ausblick

In diesem Beitrag haben wir einen Ansatz vorgestellt, mit dem sich aus formalisierten Anwendungsfällen eine konsistente visuelle Repräsentation in UML automatisch generieren lässt. Anwendungskontext ist der Übergang von informellen textuellen Repräsentationen zu Modellen und ihren verschiedenen alternativen Repräsentationen. Während wir in vorhergehenden Arbeiten [FS05, FP05] den ersten Schritt, die Formalisierung, detailliert beschrieben haben, lag der Fokus dieses Papiers auf dem zweiten Schritt, dem Generieren verschiedenster Repräsentationen.

Der Abgleich von textuellen und grafischen Repräsentationen von Anwendungsfällen zählt nach wie vor zu den aktuellen Fragestellungen in der Forschung [GLM+05]. So schließt ein idealer iterativer Softwareprozess zwar die Anforderungen mit ein, MDA-Prozesse beginnen jedoch erst mit dem Analysemodell [KWB03]. Die Erstellung des plattformunabhängigen Modells aus den textuell beschriebenen Anforderungen bleibt offen. Mit unse-

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0"
      xmlns:xmi="http://www.omg.org/XMI"
      xmlns:uml="http://www.eclipse.org/uml2/1.0.0/UML">
  ...
  <uml:CallOperationAction xmi:id="_4mAwt2_aEduZPozAIdpC6g"
    name="deleteteMessageSlot"
    incoming="_4mAwvW_aEduZPozAIdpC6g"/>
  <uml:DecisionNode xmi:id="_4mAwu2_aEduZPozAIdpC6g"
    name="Slot.full()"
    outgoing="_4mAwvW_aEduZPozAIdpC6g
              _4mAwvm_aEduZPozAIdpC6g"
    incoming="_4mAwtW_aEduZPozAIdpC6g"/>
  <uml:MergeNode xmi:id="_4mAwvG_aEduZPozAIdpC6g"
    name="endunnamed Switch"
    outgoing="_4mAwtm_aEduZPozAIdpC6g"/>
  <uml:ControlFlow xmi:id="_4mAwvW_aEduZPozAIdpC6g"
    source="_4mAwu2_aEduZPozAIdpC6g"
    target="_4mAwt2_aEduZPozAIdpC6g"/>
  ...
</xmi:XMI>
```

Abbildung 6: Ausschnitt aus der XMI-Repräsentation des generierten UML-Modells

rem Ansatz zeigen wir eine Möglichkeit auf, diesen Übergang zumindest in eine Richtung werkzeuggestützt zu vollziehen.

Mit einer den *Action Words* [Buw99] ähnlichen Methode formalisieren wir in Textform vorliegende Anwendungsfallbeschreibungen und bauen interaktiv ein formalisiertes Anwendungsfallmodell auf, welches durch ein Metamodell definiert ist. Im Gegensatz zu anderen Use-Case-Metamodellen, wie z. B. [Wil01], haben wir unser Metamodell unabhängig von dem UML-Metamodell definiert, wobei wir Wert auf eine kanonische Repräsentation anwendungsfallspezifischer Konzepte und des Kontrollflusses gelegt haben. Daher weist unser UCV-Metamodell auch Ähnlichkeiten hinsichtlich Abstraktionsgrad und Begrifflichkeiten mit Metamodellen zur Repräsentation von Programmiersprachen auf. Eine alternative Möglichkeit, ein formalisiertes Anwendungsfallmodell aufzubauen, wäre der Einsatz von schablonenbasierten Editoren. Diese werden vom Nutzer jedoch oft als zu einschränkend empfunden.

Aus dem formalisierten Anwendungsfallmodell generieren wir automatisch mittels Modelltransformation eine konsistente Repräsentation als UML-Aktivitätsdiagramm. Manuelle Schritte sind dabei nur bei der Handhabung der externen UML-Werkzeuge notwendig, z. B. beim Import und Layout. Prinzipiell sind diese Schritte jedoch auch automatisierbar. Bei der Diagrammgenerierung bestehen vielfältige Erweiterungsmöglichkeiten: Einerseits ist das Generieren weiterer Diagrammarten denkbar, wie Anwendungsfalldiagramme oder Sequenzdiagramme. Andererseits lassen sich so auch verschiedene Abbildungen von formalisierten Anwendungsfällen auf eine Diagrammart integrieren, um beispielsweise die

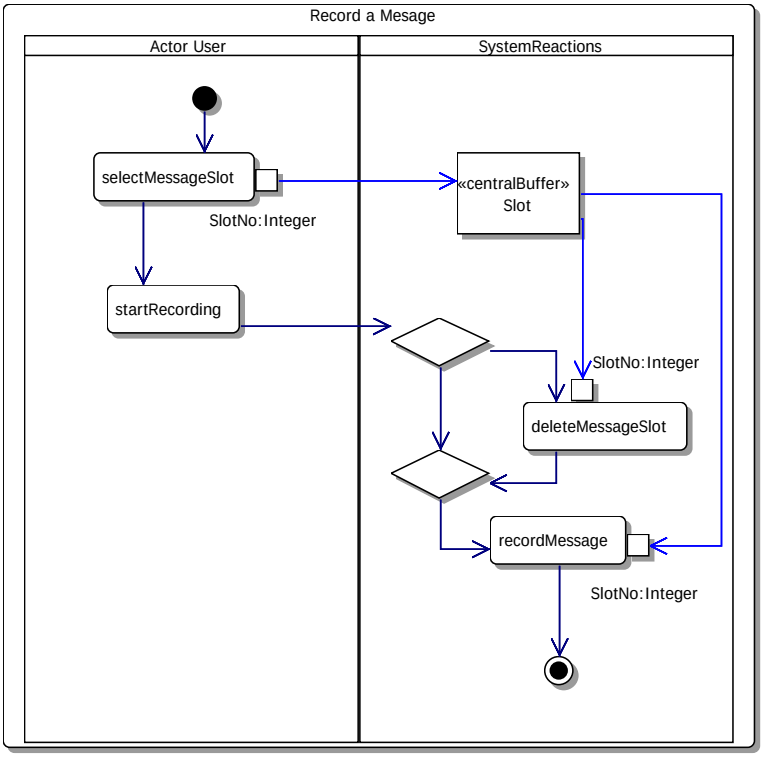


Abbildung 7: Ergebnis der Transformation ist die Visualisierung als UML-Aktivitätsdiagramm

fehlende Unterstützung spezieller UML-Konstrukte in einem Modellierungswerkzeug zu umgehen.

Durch den von uns gewählten Ansatz einer metamodellbasierten Transformation ist es möglich, auch Anwendungsfälle mit komplexeren Sprachkonstrukten zu berücksichtigen. Es ist weitere Forschungsarbeit notwendig, um die Möglichkeiten und Grenzen dieser Technik zu ermitteln.

Literatur

[ATL05] ATLAS Group, LINA & INRIA Nantes. ATL Transformation Description Template, Version 0.1, Dezember 2005.

[BEG+03] Frank Budinsky, Ray Ellersick, Timothy J. Grose, Ed Merks und David Steinberg. *Eclipse Modeling Framework.* The Eclipse Series. Addison-Wesley, 2003.

[Bor] Borland. Together 2006 Release 2 for Eclipse. http://www.borland.com/together/.

[Buw99] Hans Buwalda. Testing with Action Words. In Mark Fewster und Dorothy Graham, Hrsg., *Software Test Automation*, Kapitel 22, Seiten 446–464. Addison-Wesley, 1999.

[Ecl] Eclipse Foundation. Eclipse. http://www.eclipse.org/.

[FH06] Mario Friske und Konrad Hilse. Evaluation von Transformationsmaschinen in der modellbasierten Qualitätssicherung. Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (Band2), LNI, Bd. P-94, Oktober 2006.

[FP05] Mario Friske und Holger Pirk. Werkzeuggestützte interaktive Formalisierung textueller Anwendungsfallbeschreibungen für den Systemtest. Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (Band 2), LNI, Bd. P-68, September 2005.

[FS05] Mario Friske und Holger Schlingloff. Von Use Cases zu Test Cases: Eine systematische Vorgehensweise. In T. Klein, B. Rumpe und B. Schätz, Hrsg., *Tagungsband des Dagstuhl Workshops "Modellbasierte Entwicklung eingebetteter Systeme"(MBEES)*. Technische Universität Braunschweig, Januar 2005.

[GLM+05] Gonzalo Génova, Juan Llorens, Pierre Metz, Rubén Prieto-Díaz und Hernán Astudillo. Open Issues in Industrial Use Case Modeling. *Journal of Object Technology*, 4(6):7–14, August 2005. Special Issue: Use Case Modeling at UML-2004.

[Hil06] Konrad Hilse. Vergleich und Evaluation von Modelltransformationssprachen zur Generierung von UML-Diagrammen. Diplomarbeit, HU Berlin, August 2006.

[Hus04] Kenn Hussey. Getting Started with UML2. IBM, Juli 2004.

[JK05] Frédéric Jouault und Ivan Kurtev. Transforming models with ATL. In *Proceedings of Model Transformations in Practice Workshop*, 2005.

[KWB03] Anneke Kleppe, Jos Warmer und Wim Bast. *MDA Explained: The Model Driven Architecture - Practice and Promise*. Object Technology Series. Addison-Wesley, 2003.

[Obj03] Object Management Group. XML Metadata Interchange (XMI) Specification, Version 2.0 (formal/03-05-02). http://www.omg.org/, 2003.

[PL99] Ivan Porres Paltor und Johan Lilius. Digital Sound Recorder - A Case Study on Designing Embedded Systems Using the UML Notation. TUCS Technical Report No. 234, Turku Center for Computer Science, 1999.

[Wil01] Clay E. Williams. Toward a Test-Ready Meta-model for Use Cases. In Andy Evans, Robert France, Ana Moreira und Bernhard Rumpe, Hrsg., *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001 October 1st, 2001 in Toronto, Canada*, LNI, Bd. P-7, Seiten 270–287, Oktober 2001.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2003-03

2003-03	T.-P. Fries, H. G. Matthies	Classification and Overview of Meshfree Methods
2003-04	A. Keese, H. G. Matthies	Fragen der numerischen Integration bei stochastischen finiten Elementen für nichtlineare Probleme
2003-05	A. Keese, H. G. Matthies	Numerical Methods and Smolyak Quadrature for Nonlinear Stochastic Partial Differential Equations
2003-06	A. Keese	A Review of Recent Developments in the Numerical Solution of Stochastic Partial Differential Equations (Stochastic Finite Elements)
2003-07	M. Meyer, H. G. Matthies	State-Space Representation of Instationary Two-Dimensional Airfoil Aerodynamics
2003-08	H. G. Matthies, A. Keese	Galerkin Methods for Linear and Nonlinear Elliptic Stochastic Partial Differential Equations
2003-09	A. Keese, H. G. Matthies	Parallel Computation of Stochastic Groundwater Flow
2003-10	M. Mutz, M. Huhn	Automated Statechart Analysis for User-defined Design Rules
2004-01	T.-P. Fries, H. G. Matthies	A Review of Petrov-Galerkin Stabilization Approaches and an Extension to Meshfree Methods
2004-02	B. Mathiak, S. Eckstein	Automatische Lernverfahren zur Analyse von biomedizinischer Literatur
2005-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2005: Modellbasierte Entwicklung eingebetteter Systeme
2005-02	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part I: Stabilization
2005-03	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part II: Coupling
2005-04	H. Krahn, B. Rumpe	Evolution von Software-Architekturen
2005-05	O. Kayser-Herold, H. G. Matthies	Least-Squares FEM, Literature Review
2005-06	T. Mücke, U. Goltz	Single Run Coverage Criteria subsume EX-Weak Mutation Coverage
2005-07	T. Mücke, M. Huhn	Minimizing Test Execution Time During Test Generation
2005-08	B. Florentz, M. Huhn	A Metamodel for Architecture Evaluation
2006-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme
2006-02	T. Mücke, B. Florentz, C. Diefer	Generating Interpreters from Elementary Syntax and Semantics Descriptions
2006-03	B. Gajanovic, B. Rumpe	Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen
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 (Herausgeber)	Tagungsband Dagstuhl-Workshop MBEES 2007: Modellbasierte Entwicklung eingebetteter Systeme III