Tagungsband

Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme

Model-based Development of Embedded Systems
22. - 24.04.2009

Informatik-Bericht
2009-01
TU Braunschweig

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

Organisationskommitee

Holger Giese, Universität Potsdam

Michaela Huhn, TU Braunschweig

Ulrich Nickel, Hella KGaA Hueck&Co

Bernhard Schätz, TU München

Programmkommitee

Mirko Conrad, The Mathworks GmbH

Ulrich Epple, RWTH Aachen

Ulrich Freund, ETAS GmbH

Hardi Hungar, OFFIS

Henning Kleinwechter, carmeq GmbH

Oliver Niggemann, Hochschule Ostwestfalen-Lippe

Jan Philipps, Validas AG

Ralf Pinger, Siemens AG

Bernhard Rumpe, TU Braunschweig

Holger Schlingloff, Fraunhofer FIRST

Andy Schürr, TU Darmstadt

Joachim Stroop, dSPACE GmbH

Albert Zündorf, Universität Kassel

Inhaltsverzeichnis

Dagstuhl-Workshop MBEES:

Modellbasierte Entwicklung eingebetteter Systeme V

(Model-Based Development of Embedded Systems)

Wesentliche Elemente des Paradigmas der *modellgetriebenen Entwicklung* werden inzwischen bei der Entwicklung eingebetteter Software gelebt: So werden Modelle nicht nur zum Entwurf und zur Dokumentation von Software genutzt, sondern auch zur Generierung, Analyse, Integration, zum Test und im Betrieb, zur Wartung, Diagnose und Evolution eingesetzt; kurzum Modelle der Software, der mechatronischen und physikalischen Komponenten, aber auch der Umgebung begleiten den gesamten Lebenszyklus eines technischen Systems. Kamen beim ersten MBEES-Workshop im Jahr 2005 domänenspezifische Modellierungswerkzeuge in einzelnen Phasen der Prototypentwicklung zur Anwendung, so bilden heute Standards wie AUTOSAR eine Basis für die Durchgängigkeit von Methoden und die Kompatibilität von Werkzeugen.

Dennoch ergeben sich zwischen der Vision einer durchgängigen modellgetriebenen Entwicklung mit disziplinübergreifenden Methoden und integrierten Werkzeugumgebungen und der Pragmatik von XML-Austauschformaten eine Reihe spannender Fragen, die in den Beiträgen des diesjährigen Workshops adressiert werden: Die Integration des AUTOSAR-Standards in einen werkzeuggestützten Entwicklungsprozess ist eines der drängendsten Themen. Die Erweiterung von Modellen um sicherheitsrelevante Aspekte wie Zuverlässigkeit, die Ableitung von Diagnosemodellen und Sicherheitsnachweise sind Themen, die den ganzheitlichen Anspruch der modellgetriebenen Entwicklung verdeutlichen. Automatische Modellanalyse zur Qualitätssicherung, insbesondere modellbasiertes Testen und Verifikation sind weiterhin von Interesse. Zusätzlich beobachten wir in diesem Jahr ein erhöhtes Interesse aus der Automatisierungstechnik für die modellbasierte Software-Entwicklung.

Ausgangspunkt der MBEES-Workshop-Reihe war und ist die Feststellung, dass sich die Modelle, die in einem modellbasierten Entwicklungsprozess eingesetzt werden, an der Problem- anstatt der Lösungsdomäne orientieren müssen. Dies bedingt einerseits die Bereitstellung anwendungsorientierter Modelle (z.B. MATLAB/Simulinkartige für regelungstechnische Problemstellungen, Statechartartige 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. Geeignete Modellarten 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 Modellarten und ihre Werkzeuge werden die Anwendung 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 im Entwicklungsprozess, der Behandlung von Produktlinien und der Evolution langlebiger Modelle. Die in diesem Tagungsband zusammengefassten Papiere stellen sowohl gesicherte Ergebnisse, als auch 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 vier, in den Jahren 2005 bis 2008 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. In diesem Jahr beobachten wir eine erfreuliche Zunahme an Beiträgen aus den Domänen Automatisierungstechnik und Eisenbahntechnik, die den traditionell stark vertretenen automotiven Bereich ergänzen. Die Beiträge adressieren verschiedenste Aspekte modellbasierter Entwicklung eingebetteter Softwaresysteme, unter anderem:

- Domänenspezifische Ansätze zur Modellierung von Systemen

- Durchgängiger Einsatz von Modellen

- Modellierung und Analyse spezifischer Eigenschaften eingebetteter Systeme (z.B. Echtzeit- und Sicherheitseigenschaften)

- Konstruktiver Einsatz von Modellen (Generierung)

- Modellbasierte Validierung, Verifikation und Diagnose

- Evolution von Modellen

Das Organisationskomitee ist der Meinung, dass mit den Teilnehmern aus Industrie, Werkzeugherstellern und der Wissenschaft die bereits seit 2005 erfolgte Community-Bildung erfolgreich weitergeführt wurde, und der MBEES Workshop demonstriert, dass eine solide Basis zur Weiterentwicklung des Themas 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 April 2009,

Das Organisationskomitee:

Holger Giese, Univ. Potsdam

Michaela Huhn, TU Braunschweig

Ulrich Nickel, Hella KGaA Hueck&Co

Bernhard Schätz, TU München

Mit Unterstützung von

Axel Zechner, TU Braunschweig

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. SSE ist auch Mitherausgeber des Journals on Software and Systems Modeling.

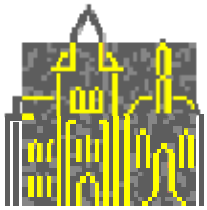
Der Lehrstuhl für Software Systems Engineering der TU München entwickelt in enger Kooperation mit industriellen Partnern modellbasierte Ansätze zur Entwicklung eingebetteter Software. Schwerpunkte sind dabei die Integration ereignisgetriebener und zeitgetriebener Systemanteile, die Berücksichtigung sicherheitskritischer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

Besonderer Fokus der Arbeit des Fachgebiets "Systemanalyse und Modellierung" des Hasso Plattner Instituts liegt im Bereich der modellgetriebenen Softwareentwicklung für softwareintensive Systeme. Dies umfasst die UML-basierte Spezifikation von flexiblen Systemen mit Mustern und Komponenten, Ansätze zur formalen Verifikation dieser Modelle und Ansätze zur Synthese von Modellen. Darüber hinaus werden Transformationen von Modellen, Konzepte zur Codegenerierung für Struktur und Verhalten für Modelle und allgemein die Problematik der Integration von Modellen bei der modellgetriebenen Softwareentwicklung betrachtet.

Als international tätiger Automobilzulieferer begründet die Hella KGaA Hueck & Co. ihre globale Wettbewerbsfähigkeit aus einer klaren Qualitätsstrategie, einem weltweitem Netzwerk und der Innovationskraft der rd. 25.000 Mitarbeiterinnen und Mitarbeiter. Licht und Elektronik für die Automobilindustrie und automobile Produkte für Handel und Werkstätten sind unsere Kerngeschäftsfelder. Mit einem Umsatz von 3,7 Milliarden Euro im Geschäftsjahr 2006/2007 ist unser Unternehmen ein weltweit anerkannter Partner der Automobilindustrie und des Handels.

Die Validas AG ist ein Beratungsunternehmen im Bereich Software-Engineering für eingebettete Systeme. Die Validas AG bietet Unterstützung in allen Entwicklungsphasen, vom Requirements-Engineering bis zum Abnahmetest. Die Auswahl und Einführung qualitätssteigender Maßnahmen folgt dabei den Leitlinien modellbasierter Entwicklung, durchgängiger Automatisierung und wissenschaftlicher Grundlagen.

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.



Schloss Dagstuhl wurde 1760 von dem damals regierenden Fürsten Graf Anton von Öttingen-Soetern-Hohenbaldern erbaut. Nach der französischen Revolution und der Besetzung durch die Franzosen 1794 war Dagstuhl vorübergehend im Besitz eines Hüttenwerkes in Lothringen. 1806 wurde das Schloss mit den zugehörigen Ländereien von dem französischen Baron Wilhelm de Lasalle von Louisenthal erworben. 1959 starb der Familienstamm der Lasalle von Louisenthal in Dagstuhl mit dem Tod des letzten Barons Theodor aus. Das Schloss wurde anschließend von den Franziskus-Schwestern übernommen, die dort ein Altenheim errichteten. 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.

Model Driven Automation Engineering – Characteristics and Challenges

Michael Schlereth[1], Sebastian Rose[2], Prof. Dr. Andy Schürr[2]

[1]Industry Sector, Industry Automation & Drive Technologies Divisions,
I IA&DT ATS 41, Siemens AG,
Gleiwitzerstraße 555, D-90475 Nürnberg
michael.schlereth@siemens.com

[2]Real-Time Systems Lab, Data Systems Technology Institute,
Electrical Engineering & Information Technologies, Darmstadt University of Technology
Merckstraße 25, D-64283 Darmstadt
{sebastian.rose|andy.schuerr}@es.tu-darmstadt.de

Abstract: Model Driven Software Development (MDSD) uses precisely defined domain specific models that are transformed into executable code by a sequence of model transformations. In this paper we present the research activities planned in year 2009 by Real-Time Systems Lab, Darmstadt University of Technology, together with Siemens Industry, Nuernberg, that will investigate the applicability of MDSD concepts within the domain of automation engineering for production systems called Model Driven Automation Engineering (MDAE). A comparison of MDSD and MDAE characteristics points out our main working topics. We also present an application scenario, which will be used to demonstrate the MDAE usage in practice.

1 Introduction

Mechatronic engineering is about integration of different engineering disciplines, mainly mechanical engineering, electrical engineering, and software engineering (information technology). Within the machine and plant engineering process, software engineering is part of automation engineering. Automation engineering deals with configuration and programming of devices like programmable logic controllers (PLC), motion controllers, and human machine interface (HMI) panels. Additional minor engineering disciplines are pneumatic engineering and hydraulic engineering. Each discipline follows its own design methodology and uses specific engineering models. Within the system development process, the sub-processes of the mechatronic engineering disciplines run in parallel with their own design iterations and design workflows. Each discipline has a set of mainstream design tools for different types of models, different design principles and a way of thinking evolved in the past, depending on the maturity of a specific discipline.

Automation engineering, the key activity within model driven automation engineering, requires the integration of information from the other mechatronic engineering disciplines (see figure 1). Regarding the running automation system at the end, each discipline has a focus on specific aspects and parts. The final machine or plant is a complex system with a lot of interacting parts created by engineers of different disciplines.

Figure 1: Mechatronics – synergy from the interaction of different disciplines [VD04a]

The degree of adoption of modeling heavily varies within the engineering disciplines:

- Mechanical engineering uses form oriented models e.g. with feature based design in 3D construction. Function oriented construction is widely used in plant engineering (e.g. with reference designation), but less popular in machine engineering.
- Electrical Engineering uses location and product models to define the wiring and to add the automation devices to the bill of material (BOM).
- Within Automation engineering modeling is not well-established. Modeling is only used for specific tasks such as production process models for virtual commissioning or control loop design.

The objective of model driven automation engineering is to promote the usage of models within automation engineering and to establish information exchange and integration mechanisms based on model transformation techniques between the mechatronic engineering disciplines. Enabling model usage within automation engineering does not necessarily mean that models must be explicitly introduced as elements of automation engineering. In a first step, models that are already available within the automation engineering system such as signals or hardware device configuration should be exposed to the mechatronic workflow. These exposed design artifacts can be used for assisted data exchange between discipline specific tools. By following steps, model elements that are essential for traceability of design artifacts within the mechatronic design workflow – such as the functional structure – can be added to the current automation engineering systems.

The purpose of this paper is the presentation of the research activities planned for 2009 by Real-Time Systems Lab, TU Darmstadt, and Siemens Industry, Nuernberg, concerning model driven automation engineering (MDAE). The next chapter “usage scenario” introduces the environment that will be used to prove the usefulness of model driven automation concepts. The commonalities and differences between model driven engineering in general and model driven automation engineering are explained in chapter “MDAE in relation to MDA and MDSD”. Finally the chapter “Working Directions” outlines the working packages that will be addressed within the next months.

2 Usage Scenario

The MDAE environment integrates the model parts relevant for the construction of the automation solution from different engineering disciplines, domains, and tools. These models are especially the function oriented structure from mechanical engineering, the device and signal model from electrical engineering and the automation system structure and the program code from automation engineering.

From a tool perspective, MDAE does not force the user to adopt specific engineering tools or migrate to a common super-tool, but enables integration of different engineering environments by the use of model adapters and user defined transformations between these adapters. This is especially important for simulation tools that are usually specialized within their application area and are used for specific design tasks.

Machine and plant builders usually do not rely on a single source for the automation system provider, but follow a strategy with alternative system providers. Also the customer of the machine builder may demand for a specific automation solution to have a common system environment within their facilities. Therefore, MDAE supports the definition and usage of different target platforms by means of model transformations. From the perspective of the MDAE environment these different target platforms are handled in the same way as different modeling or simulation tools, which are integrated with specific adapters and transformations. From a user’s perspective the transformation to the target platform has a higher variability than other transformations since the transformation may vary e.g. depending on different coding standards or different application domains.

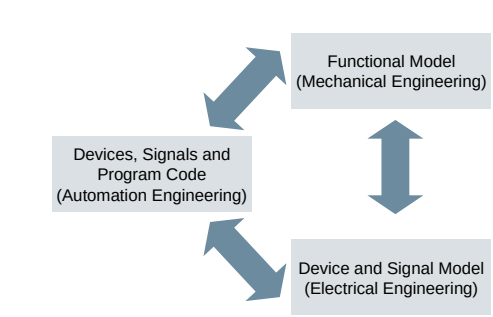


Figure 2: MDAE models within evaluation scenario

Within production machine development, not only logic controller programming, but also motion control programming must be part of the MDAE environment. For example a high-bay warehouse, which is available as a demonstration plant, is an application with both kinds of programming. The movement of the warehouse's storage access crane is controlled by motion axis while the feeding of palettes is controlled by a logic controller (see figure 3). This example also demonstrates the wide scope of automation devices that must be considered within MDAE such as binary IO, drive control, identification systems, PLC and HMI devices.

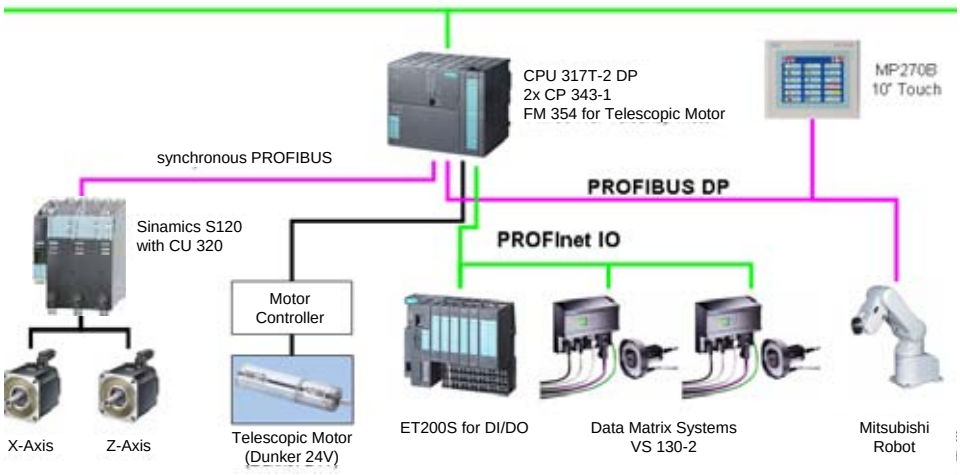


Figure 3: Automation components of a high-bay warehouse demonstration plant

3 MDAE in relation to MDA and MDSD

Since the late eighties and the introduction of Computer Aided Software Engineering (CASE) the size and complexity of software and systems is still growing [Sc06]. Model Driven Software Development (MDSD) [St07] formerly known as Model Driven Development (MDD) may be seen as a branch of the CASE tool and the Integrated Software Factory research activities of the last millennium that puts a strong emphasis on the integrated usage of models in all stages of engineering processes. Model Driven Architecture (MDA) [MM03] is a methodology for MDSD/MDD defined by the Object Management Group (OMG.) MDA usually considers a single thread of models that are assigned to three different levels of abstraction and that are used and refined over time during the development process and software lifecycle.

MDA and MDSD emphasize the use of three aspects summarized by [VB05]:

1. Meta-modeling for modeling-language design
2. Precise models
3. Automatic transformations between models

(1.) The (visual) languages for describing models within MDA are not predetermined. Although existing languages like UML can be used, domain oriented customization is necessary for many application fields. The discipline of meta-modeling enables customization in different ways. In the case of UML it is possible to build UML-Profiles as UML-PA [KV05]. Building modeling languages from scratch is possible using for example MOF [Ob04] or EMF [Sk08].

(2.) Each model within MDA has a formalized meaning according to its intention. Formalization in this case is not at the same level as formalization of mathematical models. Precise models are necessary for machine-processing. Precise means that the model adheres to a set of constraints that are a required precondition for the application of a model transformation. These constraints could for example define specific model semantics or syntax restrictions. In contrast to CASE, MDA demands separated models for technical and domain specific aspects of a system. By separation each aspect is changeable on its own.

(3.) Current implementations of transformations are mainly designed for forward generation of standard mappings. Within an iterative workflow with many small transformations in context-dependent mappings, these monolithic approaches are hard to handle. The objective of MDAE is to define bidirectional transformations on different levels of granularity. These transformations should be transparent to the user and enable a mixture of automatic and manual model modifications.

3.1 Differences between MDA and MDAE

MDA deals with one business model that is used for software development. Within MDA the meaning of the term business domain is not well-defined. Other disciplines beside software are disregarded. Mechatronics [VD04a] is concerned with three disciplines of engineering. These are mechanical, electrical, and software engineering. Each discipline on its own has well-established tools supporting domain specific modeling. Mechanical engineers, for example, use one of a variety of CAD tools for modeling the form oriented structure of a machine. Additionally mechanical behavior is used to describe the path-time diagram, which is a different kind of diagram typically known to software engineers. In comparison to mechanical and electrical engineering, software engineering is the youngest discipline. As a consequence modeling paradigms and principles are still evolving.

The definition of a platform given in [MM03] contains frameworks, programming languages, the application architecture and hardware-architecture. MDAE additionally includes platforms in the broader sense of mechanical and electrical parts with a wide range of choices for a large system (e.g. selection of drives from different vendors). Thus a platform in the context of MDAE applies to more complex and smaller pieces of hardware with a lot more of variations. Additionally, hardware is more than platforms running software. The automation system consists of mechanical parts in a large scale. These parts have physical properties like force, speed, and weight for example. These properties are part of mechanical models and do not fit into known terms of software engineering.

The lifecycle of the MDA methodology usually relies on a single linear thread of transformations in the direction of the implemented system for refinement and combination of domain specific and technical related purposes. Each MDA model on a certain level of abstraction has a unique predecessor model on a higher level of abstraction (plus some platform definition parameters that control its creation by means of transformations). Furthermore, a single (software) development process controls the iterated development and refinement of all needed models. MDAE, on the other hand, has to deal with a considerably higher degree of concurrent engineering activities of various (sub-)disciplines. As a consequence, the simultaneous evolution of different discipline specific models at the same level of abstraction with their own predecessor and successor models on other levels of abstraction plays a much more important role. Therefore, MDAE has to combine the vertical refinement of models along the line of MDA principles with the horizontal integration of different modeling threads belonging to at least three different disciplines.

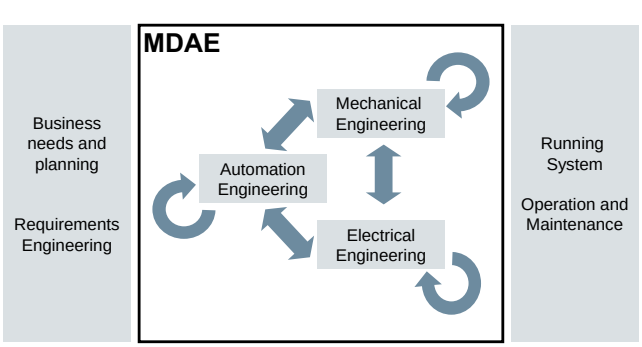


Figure 4: Simultaneous Evolution of Models

In figure 4 these three disciplines are shown within the MDAE scope. The workflows between these disciplines are not standardized but depend on involved tools, vendors or business scenario (e.g. process automation, factory automation, logistics). The model transformations, which support these different workflows, must support a wide range of transformation operations (e.g. according to the classification of [CH03]. A sequential process that excludes concurrent engineering activities, defined as part of the MDA methodology would increase time and costs. Therefore, bidirectional or even multi-directional information exchange and transformations between pairs or sets of models of cardinality n > 2 are needed [KS05].

In addition, figure 4 shows that MDAE does not claim to include business planning, requirements engineering or the running system. Business planning and requirements engineering currently do not provide exact models that could be interfaced by MDAE. On the other end of the workflow, the amount of operation data and the kind of information within the running systems requires a separation from the engineering data.

3.2 Problems and Challenges

As described above each discipline has types of models and tools concerning discipline specific needs. The working system at the end is a primary goal for all involved parties.

Hardware and software coexistence in MDAE allows a lot of solution variance. Automation systems can be realized with different allocations of hardware and software. Depending on hardware-software-partitioning, modeling has a different focus. This has an influence on the transformation needs. There is no common type of workflow with a predefined way of transforming information. Customized transformations are required for reflecting actual needs. Sophisticated modularization concepts (e.g. components) within models are strongly required at the meta-level. Both meta-models and transformations need that modularization for scalability, flexibility and manageability reasons [KKS07]. By means of modularization, existing or previously defined meta-models and transformations can be reused and refined. These capabilities enable efficient and long term handling of changes caused by different workflows, domains, tools, and project characteristics.

MDAE explicitly allows temporary inconsistency to support collaboration (between worldwide locations and between people), to support parallel workflows and to support design alternatives and studies. The strength of MDAE is its support of incremental workflows to consistency: in small iterations locally and daily for a single worker, on module and discipline level for engineering teams and on system level for milestone delivery.

The MDAE approach to consistency strongly requires sophisticated concepts for modeling in the large. Systems with a high number of different models and users with different domain knowledge and collaboration needs demand a scaling environment for modeling. Even in small organizations, the tooling environment is dynamic because for example within simulation continuously new tools with specialized features enter the market and must be integrated in the design workflow.

Therefore, the main challenges of MDAE include the (1) development of methods for reverse engineering meta-models from sometimes rather arcane COTS tool APIs and documentation that are relevant for information exchange, (2) the mapping of these meta-models to proprietary model interfaces (more precisely: modeling tool interfaces), and (3) the integration of the content behind these interfaces by means of customizable and refineable bi- or multi-directional and incremental model transformations.

4 Related Work

With respect to the automation model which is the focus for model driven automation engineering existing research activities cover reengineering of IEC 61131-3 [In03] based PLC programs and combining UML with PLC programming. [Ba06a] uses transformations from code over XML to state machines to reengineer the structure of IEC 61131-3 based PLC programs. XML to IEC 61131-3 code and XML to hardware device configurations mappings are published in [EM05] and [EMO05]. Usage of UML in automation engineering is part of several research activities e.g. [Vo08]. MDAE will consider results of these activities for building libraries and transforming existing PLC software to a PLC code model. Since MDAE does not claim to introduce new models like UML, but reuses existing models, UML is important as a meta-modeling technique to integrate the relevant model parts into the transformation environment.

Integration of mechatronic engineering models has been subject of multiple research activities in the last few years. [Ge05] shows the integration of mechatronic design models with focus on functional structure and system structure ("Wirkstruktur"). Consistency and mapping rules between models are defined by graph transformations described by story boards. The engineering process of body in white production lines in automotive industry is described by [Ki07]. [Gr06] focuses on the generation of electrical wiring and fluid diagrams. [Ba06b] uses the model of an extended functional structure for PLC code generation. Modularization and composition of mechatronic products is investigated ,[Do02], but not with respect to PLC code generation. Meta-modeling of a machine tool based on ontologies and mediators between domain specific models supports model driven development of machine tools in the work of [Le08]. Föderal [VD04b] and its successor Aquimo [Li08] assume similar to MDAE that domain specific engineering models reside in domain specific tools; but they still rely on a common base model and do not define transformation models between domains.

Model to model transformations in automation technology based on the framework OpenArchitectureWare [Op08] are described by [Ma08]. The key elements of this approach are a forward directed workflow and an explicit definition of exchangeable platforms based on templates.

The Model-Driven Engineering overview by [Sc06] describes the lack of an „integrated view" between different engineering applications. Dealing with complexity in large systems each participant has a very small focus in comparison to the size of the whole system. Changes of all types might have an impact elsewhere. These impacts cannot be handled by a focus of one of the development or management tools involved. A solution within MDAE might be a propagation chain of changes along model transformations that make the impact of local changes to the rest of the system visible to the user.

Within process automation, the integration between the discipline specific model of an infrastructure planning tool and a simulation tool based on triple graph grammars (TGG) is described by [Be07]. A key characteristic of triple graph grammars (TGG) is that "…TGGs can be used to synchronize and to maintain the correspondence of the two models, even if both of them are changed independently of each other; i. e., TGGs work incrementally." [KW07] The use of TGGs with models from different domains and with different structure as in MDAE needs to be investigated further.

5 Working Directions

The problems and aspects of dealing with different disciplines and domains given in section 3 can be tackled in different ways. One approach, not followed by MDAE, could try to establish an interdisciplinary collaboration by the definition of a new modeling language that fits all the requirements and needs of mechatronic engineering. This might be driven by tool vendors trying to establish the one and only modeling software in place. Given the lifecycle of a system, such software must support requirements analysis and co-working of different disciplines from the first requirement until the acceptance. Nowadays there is no such tool or initiative to develop such a tool known to the authors.

The primary topic of MDAE is, thus, a focus on partial tool integrations as needed by concurrent engineering within the design and construction phase of automation business.

5.1 Defining the MDAE environment

The main purpose of using model driven automation engineering (MDAE) is an enhanced quality of complex automation systems. Quality goals for model driven automation engineering in the context of MDAE are:

- Meeting the requirements (especially from mechanical construction and customers).
- Being in cost and time.
- Reducing the amount of design errors.
- Reduce project risks.
- Meet and use standards for development (internal or external standards).
- Manage complex systems.

The bidirectional transformation between models eases communication between engineering phases (e.g. process requirements between the mechanical department and basic automation engineering) and engineering domains (e.g. between automation engineering and electrical engineering). This transformation together with the reuse of existing design artifacts reduces engineering cost, engineering time and the amount of design errors.

The transformations within model driven automation engineering are used together with libraries of modules and transformations. These libraries enable the support of coding standards and reduce the overall complexity of the system by encapsulation.

The 3 key elements of MDAE are summarized in figure 5: precise models, model extensions and model transformations. MDAE integrates multiple models which are not dependent on each other in a hierarchical or timely order, but can exchange information iteratively throughout the design process. To be used within the MDAE environment, each model has an extension related to the transformation. This extension might hold additional model elements or attributes such as IDs or timestamps. Part of the extension can be additional rules that the model must adhere to. Based on these extensions, the MDAE transformations realize an information exchange between the different models. The extensions as well as the transformations do not apply to the complete models, but only to the part of the model which is relevant for the information exchange.

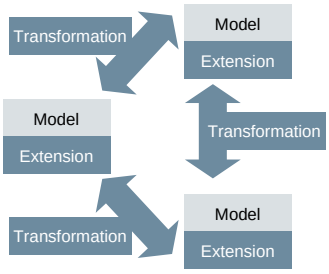


Figure 5: MDAE key elements

5.2 Course of action

The evaluation scenario will transform data between three main models as defined in the usage scenario (see figure 2):

- the functional model from mechanical engineering defines the structure and intended behavior of the machine
- the device and signal model from electrical engineering defines electrical devices and the signals attached to these devices
- within automation engineering, the controller engineering tools configure devices, assigns signals and program code

Although the model integration environment shown in figure 5 does not favor one of the models over another model, the focus of the work within model driven automation engineering is set on the automation engineering model. Special efforts within the automation engineering model are put on domain modeling of the automation system structure, on PLC code reengineering and PLC software engineering. These efforts are justified by the growing number of features that are implemented in software and not in mechanics as well as by the low use of modeling techniques within automation engineering in contrast to other disciplines.

The transformations will work on MOF [Ob08] compliant interfaces to access the model elements relevant for the transformation. At the beginning there will be small and specialized meta-models of one tool concerned with information relevant for transformations to other models. Each of these meta-models only contains the concepts required for a transformation. In contrast, a full-featured meta-model would include all concepts given by the corresponding tool.

The transformations will be defined by triple graph grammars [Sc94]. The main focus is on working transformations. The outcome from the information exchange will be examined running different examples. As already mentioned each tool model is accessible through MOF compliant interfaces in terms of an adapter. This code is generated from specialized meta-models. Before a transformation may take place, the models involved have to be in a valid state according to their meta-model and model extensions. Due to specialized meta-models this is a reduction of all instances allowed by the tool. Comparable approaches are modeling or programming guidelines aiming at higher quality and interchangeability as e.g. [PL08].

5.3 Future Work

As described before, information will be exchanged between models as pairwise integration based on meta-models. For each engineering discipline (mechanical, electrical and automation engineering, shown as circles in figure 6) modeling information that is relevant for model transformations between engineering disciplines is described by a meta-model. Model driven automation engineering assumes that these model transformations are not monolithic, but small and incremental transformations that can be executed in many iterations. Therefore, in a first step these model transformations need not operate on a general meta-model for each engineering discipline, but can use transformation specific meta-models (shown as squares in figure 6). These different meta-models could for example reflect different tools that are used within an engineering discipline.

For a growing repository of involved tools, refactoring and merging of different meta-models might take place over time. As a result of this merging process of specialized meta-models, discipline-specific concepts from a collection of tools will result in one meta-model. One model defined by such a meta-model should be distributed among multiple tools established in the specific discipline. In two further steps this could potentially be merged into one big meta-model for mechatronic engineering in the same way (right side of figure 6). For reasons of simplicity, not shown in figure 6 are the potentially different abstraction levels of the integrated meta-models. Discipline specific meta-models (e.g. mechanical, electrical, and automation engineering) share information on the same abstraction level. For example, the integration of domain specific modeling (e.g. process modeling for production processes within basic engineering) with an implementation model (e.g. manufacturing control of a programmable logic controller within detailed engineering) links different abstraction levels.

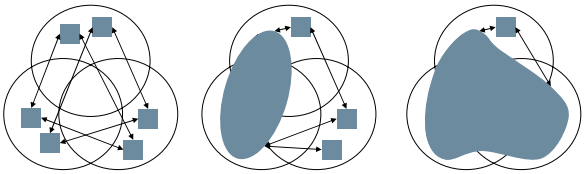


Figure 6: Evolution of Meta-Models

6 Conclusion

Within model driven automation engineering the focus of research activities planned in year 2009 will be on bi-directional integration of automation system related engineering models. MDAE does not force the user to integrate existing engineering discipline specific models in a common “super or reference model” for all disciplines, but integrates partial model views of discipline specific engineering tools by bi-directional model transformations. Discipline specific tools are interfaced by Meta Object Facility (MOF) compliant adapters, which expose automation system relevant parts of the discipline specific model. Transformations are defined by triple graph grammars (TGG). The main challenges for model adapters and transformations are definitions of model associations that are understandable and readable to the end-user. Additionally, data exchange needs to be transparent and testable, to ensure successful and productive use of model driven automation by machine and plant builders.

The MDAE approach thus most offers sophisticated (meta-) tool support for systematic and efficient:

- reverse engineering of meta-models from existing engineering tools relevant for automation engineering (independent of the abstraction level of these meta-models)
- development of modeling tool adapters that realize partial (overlapping) transformation specific views on models inside tools
- restriction and customization of tool adapters and their meta-models by means of additional integrity constraints
- realization of model extensions that hold additional integration specific data

Model driven automation engineering thus raises the quality of automation system development processes and products. Assisted data exchange and reuse of design artifacts, saves time and reduces the number of errors.

Beyond the current working focus other research activities around model driven automation engineering will cover integration of simulation, detection and repair of model inconsistencies, collaboration support and integration in product lifecycle management (PLM) systems.

7 References

[Ba06a] Bani Younis, M.: Re-engineering approach for PLC programs based on formal methods. Re-Engineering-Ansatz für SPS-Programme auf Basis formaler Beschreibungen. Dissertation. Shaker, Aachen, 2006.

[Ba06b] Bathelt, J.: Entwicklungsmethodik für SPS-gesteuerte mechatronische Systeme. Dissertation, Zürich, 2006.

[Be07] Becker, S.M. et. al.: A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. In: Software and Systems Modeling(3), 2007; S. 287–315.

[CH03] Czarnecki, K.; Helsen, S.: Classification of Model Transformation Approaches. OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[Do02] Dohmen, W.: Interdisziplinäre Methoden für die integrierte Entwicklung komplexer mechatronischer Systeme. Dissertation. München, Utz, 2002.

[EMO05] Estevé, D.; Marcos, M.; Orive, D.: Modelado formal de sistemas de control distribuido y adaptación a diferentes fabricantes, 2005, http://www.disa.bi.ehu.es/eestevez/publications/downloads/JJTR05.pdf.

[EM05] Estévez, E.; Marcos, M.: IEC 61131-3 code generation using XML technologies, 2005, http://www.disa.bi.ehu.es/eestevez/publications/downloads/CEDI05.pdf.

[Ge05] Gehrke, M.: Entwurf mechatronischer Systeme auf Basis von Funktionshierarchien und Systemstrukturen. Dissertation, Paderborn, Oktober 2005.

[Gr06] Grätz, F.M.: Teilautomatische Generierung von Stromlauf- und Fluidplänen für mechatronische Systeme. Dissertation. Utz Herbert, München, 2006.

[In03] International Electrotechnical Commission (IEC): Programmable controllers – Part 3: Programming languages. 2.0. Auflage(IEC 61131-3), 2003-01.

[KV05] Katzke, U.; Vogel-Heuser, B.: UML-PA as an engineering model for distributed process automation. In (Piztek, P. Hrsg.): Proceedings of the 16th IFAC world congress. Prague, Czech Republic, July 3 - 8, 2005. Elsevier, Oxford, 2005.

[Ki07] Kiefer, J.: Mechatronikorientierte Planung automatisierter Fertigungszellen im Bereich Karosserierohbau. Dissertation. Univ. des Saarlandes Lehrstuhl für Fertigungstechnik, Saarbrücken, 2007.

[KW07] Kindler, E.; Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios, June 2007.

[KKS07] Klar, F.; Königs, A.; Schürr, A.: Model Transformation in the Large. Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, New York: ACM Press, 2007; ACM Digital Library Proceedings, 285-294., New York, 2007.

[KS05] Königs, A.; Schürr, A.: Multi-Domain Integration with MOF and extended Triple Graph Grammars. In (Bezivin, J.; Heckel, R. Hrsg.): Language Engineering for Model-Driven Software Development, 2005.

[Le08] Lercher, B.: Konzeption und System einer Integrationsplattform zur Entwicklung von Werkzeugmaschinen. Dissertation, München, 07.01.2008.

[Li08] Litto, M.: AQUIMO, 2008, http://www.aquimo.org/.

[Ma08] Maurmaier, M.: Modell-zu-Modell-Transformationen in der Automatisierungstechnik. In: Automation 2008. Lösungen für die Zukunft. VDI Berichte 2032. VDI-Verl., Düssdeldorf, 2008.

[MM03] Mukerji, J.; Miller, J.: MDA Guide Version 1.0.1, 2003, http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.

[Ob04] Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification, 15.10.2004, http://www.omg.org/docs/ptc/04-10-15.pdf.

[Ob08] Object Management Group: OMG's MetaObject Facility (MOF) Home Page, 18.11.2008, http://www.omg.org/mof/.

[Op08] OpenArchitectureWare: openArchitectureWare.org - Official openArchitectureWare Homepage. The leading platform for professional model-driven software development, 2008, http://www.openarchitectureware.org/.

[PL08] PLCOpen: PLCopen Home, 20.11.2008, http://plcopen.org/.

[Sc06] Schmidt, D.C.: Model-Driven Engineering. In: IEEE Computer 39(02), 2006, http://www.cs.wustl.edu/~schmidt/GEI.pdf.

[Sc94] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Heidelberg: Springer Verlag, 1994; Lecture Notes in Computer Science (LNCS), Vol. 903, 151-163., 1994.

[Sk08] Skrypuch, N.: Eclipse Modeling - EMF - Home, 2008, http://www.eclipse.org/modeling/emf/.

[St07] Stahl, T. et. al.: Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management. 2., aktualisierte und erw. Aufl, dpunkt, Heidelberg, 2007.

[VB05] Vanhooff, B.; Berbers, Y.: Breaking Up the Transformation Chain. OOPSLA 2005 Workshop "Best Practices for Model Driven Software Development", 2005, http://www.softmetaware.com/oopsla2005/vanhooff.pdf.

[VD04a] VDI-Gesellschaft Entwicklung Konstruktion Vertrieb: Entwicklungsmethodik fuer mechatronische Systeme(VDI 2206), 2004-06-01.

[VD04b] Baukastenbasiertes Engineering mit Föderal. Ein Leitfaden für Maschinen- und Anlagenbauer. VDMA Verl., Frankfurt am Main, 2004.

[Vo08] Vogel-Heuser, B.: Automation & embedded systems. Oldenbourg Industrieverl., München, 2008.

Modellgetriebene Entwicklung von Automatisierungssystemen

Mathias Maurmaier, Peter Göhner

Universität Stuttgart
Institut für Automatisierungs- und Softwaretechnik
Pfaffenwaldring 47 - 70550 Stuttgart
mathias.maurmaier@ias.uni-stuttgart.de

Abstract: Die Komplexität moderner Automatisierungssysteme nimmt stetig zu. Die modellgetriebene Entwicklung hat das Potenzial, diese Komplexität beherrschbar zu machen, und gleichzeitig die Effizienz in der Entwicklung und die Qualität der Entwicklungsergebnisse zu steigern. Doch hohe Hardware-Software-Abhängigkeiten, verschiedenartige Anforderungen, die bei der Entwicklung berücksichtigt werden müssen und die Vielzahl unterschiedlicher Modellierungssprachen stellen spezifische Herausforderungen an ein modellgetriebenes Vorgehen in der Automatisierungstechnik. Im Folgenden wird ein Konzept der modellgetriebenen Systementwicklung vorgestellt, das diese Herausforderungen berücksichtigt und somit der Automatisierungstechnik die Vorteile der modellgetriebenen Entwicklung eröffnet.

1. Herausforderungen bei der Entwicklung von Automatisierungssystemen

Automatisierungssysteme sind komplexe Hardware-Software-Systeme, deren Ziel die Führung und Überwachung eines technischen Prozesses ist. Bei der Entwicklung derartiger Systeme sind verschiedene Disziplinen wie Softwaretechnik, Hardwareentwicklung oder Energietechnik beteiligt. Ein wichtiger Hebel zur Beherrschung der Komplexität bei der Entwicklung liegt in der Modellierung [ZVEI06]. Daher werden zur Systementwicklung viele unterschiedliche Modelle genutzt. Da Abhängigkeiten zwischen den einzelnen Modellen nicht automatisiert verwaltet werden, ist ein hoher Aufwand für manuelle Mehrfacheingaben von Informationen und für die Sicherstellung der Konsistenz der Modelle erforderlich [Sch08].

Die Hardware eines Automatisierungssystems bestehend aus Sensoren, Aktoren und Recheneinheiten sowie die Software sind sehr stark integriert, was zu einer Vielzahl von Abhängigkeiten bei der Entwicklung führt. Des Weiteren werden bei der Entwicklung eines Automatisierungssystems wo möglich mehrfach verwendbare Teillösungen eingesetzt. Diese Teillösungen bestehen aus Hardware und einer Spezifikation oder Implementierung der zugehörigen Software. Eine Anforderung der Automatisierungstechnik an die Entwicklungsmethode ist daher, mehrfach verwendbare Teillösungen bereitzustellen und Hardware-Software-Abhängigkeiten zu berücksichtigen.

Bei der Entwicklung eines Automatisierungssystems müssen unterschiedliche Anforderungen erfüllt werden. Primär muss der technische Prozess geführt und überwacht werden. Weitere spezifische Anforderungen an das Automatisierungssystem ergeben sich aus der Auslegung des technischen Systems, vorhandenen Altsystemen sowie rechtlichen und wirtschaftlichen Randbedingungen. Diese Anforderungen variieren zwischen verschiedenen Automatisierungssystemen, die den gleichen technischen Prozess realisieren. Eine für die Automatisierungstechnik geeignete Entwicklungsmethode muss es ermöglichen, diese verschiedenartigen Anforderungen zu berücksichtigen.

Im Gegensatz zur Softwaretechnik konnte sich in der Automatisierungstechnik bisher keine universelle Modellierungssprache wie beispielsweise die UML etablieren. Ursache hierfür sind verschiedene an der Entwicklung beteiligte Disziplinen wie Hardware- und Softwareentwicklung sowie Domänen- und Sicherheitsexperten, die jeweils spezifische, auf ihre Bedürfnisse zugeschnittene Modellierungssprachen nutzen. Die Entwicklungsmethode muss daher offen für die Nutzung verschiedener Modellierungssprachen sein.

Neben den bisher erläuterten Anforderungen muss eine Entwicklungsmethode in der Automatisierungstechnik eine effiziente Durchführung der Entwicklung ermöglichen.

In diesem Beitrag wird, ausgehend von einer entwicklungstheoretischen Betrachtung der Systementwicklung und der klassischen modellgetriebenen Entwicklung in Kapitel 2, analysiert, warum die modellgetriebene Entwicklung in der Automatisierungstechnik bisher kaum eingesetzt wird. In Kapitel 3 wird ein Konzept der modellgetriebenen Entwicklung vorgestellt, das die spezifischen Herausforderungen bei der Entwicklung von Automatisierungssystemen erfüllt. In Kapitel 4 wird anhand der Domäne Waschautomat aufgezeigt, wie dieses Konzept in der Praxis realisiert werden kann.

2. Klassisches Entwicklungsvorgehen

2.1 Entwicklungstheoretische Betrachtung der klassischen Systementwicklung

Unter Entwicklung versteht man nach [SmBr93] die Erstellung eines Systems oder Artefakts zur Lösung eines gegebenen Problems. Wie Abbildung 1a zeigt, werden hierzu Repräsentationen der realen Welt eingesetzt. In der Repräsentation des Problems R(P) wird das zu lösende Problem mit Begriffen, Konzepten und Metriken des Problemraums beschrieben. Aus dieser Repräsentation wird über den Verlauf der Entwicklung schrittweise eine Repräsentation der Lösung R(L) erstellt, auf Basis derer das System produziert werden kann. Die Repräsentationen können unterschiedliche Formalisierungsgrade aufweisen. Eine textuelle Anforderungsspezifikation ist beispielsweise eine informale Repräsentation des zu lösenden Problems, der Quellcode der Automatisierungssoftware eine semi-formale Repräsentation der Lösung.

Das zu entwickelnde System besteht aus einer Menge von Einzelelementen, die so aufeinander bezogen sind und miteinander wechselwirken, dass sie ein gemeinsames Ziel erfüllen [Som07], d. h. ein spezifisches Problem lösen.

Die Eigenschaften des Systems ergeben sich aus den Eigenschaften der Einzelelemente und den Eigenschaften, die durch das Wechselwirken der Elemente entstehen. Letztere werden globale Systemeigenschaften genannt. Im einfachsten Fall beziehen sich alle Anforderungen, die von einem zu entwickelnden System erfüllt werden müssen, auf das zu lösende Problem. Anforderungen können sich jedoch auch auf einzelne Systemelemente oder globale Systemeigenschaften beziehen. Diese geforderten Eigenschaften schränken die Anzahl der möglichen Lösungen des Problems ein.

Im Rahmen der schrittweisen Erstellung der Repräsentation der Lösung R(L) werden eine Vielzahl von Entwurfsentscheidungen getroffen, die die Lösung beeinflussen. Diese Entscheidungen trifft der Ingenieur auf Basis seines Wissens, der geforderten Eigenschaften, der verfügbaren Technologien und der vorhandenen wiederverwendbaren Teillösungen. Bei der Entscheidungsfindung muss der Ingenieur sämtliche Abhängigkeiten zwischen den eingesetzten Technologien und Teillösungen berücksichtigen. Gibt es eine Modifikation im Ziel des Systems, den geforderten Eigenschaften oder der eingesetzten Teillösungen, müssen die Entscheidungen überprüft und eventuell korrigiert werden.

Ziel der modellgetriebenen Entwicklung ist, die Repräsentation der Lösung R(L) aus der Repräsentation des Problems R(P) automatisiert zu erzeugen. Die Grundlagen hierzu werden im folgenden Abschnitt dargestellt.

2.2 Entwicklungstheoretische Betrachtung der modellgetriebenen Entwicklung

Die modellgetriebene Entwicklung nutzt Transformationen zur automatisierten Überführung der Repräsentation des Problems in die Repräsentation der Lösung. Voraussetzung hierfür ist eine Formalisierung der Einflussfaktoren. Die formalisierte Repräsentation des Problems wird als fachliches Modell M(P) bezeichnet. Dieses Modell weist eine hohe Abstraktion und eine große Problemnähe auf. Wie in Abbildung 1b ersichtlich wird aus dem fachlichen Modell M(P) durch eine oder mehrere sequenzielle Transformationen das detaillierte Modell der Lösung M(L) generiert. Eine Transformation ist dabei die formalisierte Überführung eines Quellmodells in ein Zielmodell.

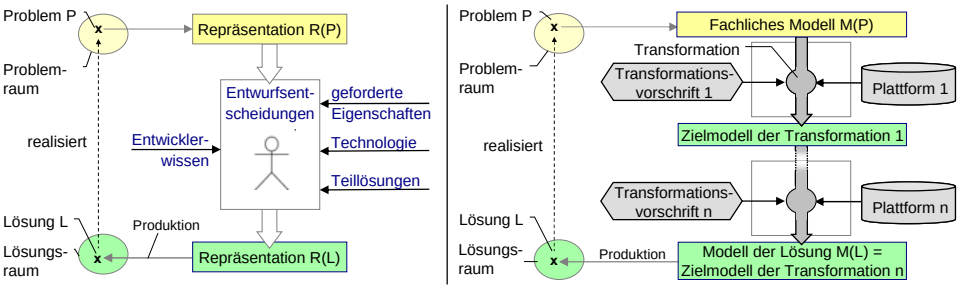


Abbildung 1: Entwicklungstheoretische Betrachtung der klassischen Systementwicklung (a) und der modellgetriebenen Entwicklung (b)

Zur Realisierung einer Transformation werden eine Plattform und eine Transformationsvorschrift benötigt. Die Plattform enthält wiederverwendbare Teillösungen einer Abstraktionsebene, deren Schnittstellen und Eigenschaften formalisiert beschrieben sind.

Die Transformationsvorschrift formalisiert den Teil des Entwicklerwissens, der zur Auswahl, Verknüpfung und Konfiguration der Teillösungen der zugehörigen Plattform benötigt wird, um die im Quellmodell modellierbaren Probleme zu lösen. Jede Transformationsvorschrift besteht aus einzelnen Transformationsregeln. Bei der Durchführung einer Transformation verknüpfen die Transformationsregeln das im Quellmodell modellierte Problem mit den Teillösungen der Plattform. Jede Transformationsregel besteht daher aus zwei Teilen [CzHe03]. Der erste Teil dient zur Identifikation und Extraktion von Informationen aus dem Quellmodell. Der zweite Teil nutzt die extrahierten Informationen und beschreibt, welche Eigenschaften die zu selektierende Teillösung haben muss, wie mehrere Teillösungen zu verknüpfen oder zu konfigurieren sind.

Da die Transformationsvorschrift die Modellelemente des Quellmodells mit den Teillösungen aus der Plattform verknüpft, ist sie von der Modellierungssprache des Quellmodells und der Beschreibung der Teillösungen abhängig. Standards zur Spezifikation der Transformationsvorschriften wie QVT [QVT08] definieren den Aufbau der Vorschriften daher mithilfe von Metamodellen. Sie sind sehr generisch und erfordern einen hohen Aufwand bei der konkreten Definition von Transformationsvorschriften im Rahmen der Einführung der modellgetriebenen Entwicklung in einer Domäne. Daher existieren zumeist in Entwicklungsumgebungen integrierte Realisierungen für spezifische Modellierungssprachen und Anwendungsdomänen der Softwareentwicklung.

Werden neben den Anforderungen, die das zu lösende Problem beschreiben, spezifische weitere Systemeigenschaften wie zum Beispiel eine hohe Zuverlässigkeit oder ein einzusetzendes Betriebssystem gefordert, so müssen durch den Entwickler die Plattformen ausgewählt werden, die diese Eigenschaften erfüllen.

Die Festlegung der benötigten Transformationen und der eingesetzten Modellierungssprachen innerhalb einer Domäne wie auch die Entwicklung der Plattformen und Transformationsvorschriften findet in einer vorgelagerten, projektunabhängigen Entwicklungsphase, der sogenannten Infrastrukturentwicklung statt. In dieser Phase wird ein Domain Engineering betrieben, um die Artefakte bereitzustellen, die später in der Systementwicklung für verschiedene Projekte mehrfach verwendet werden.

3. Modellgetriebene Entwicklung in der Automatisierungstechnik

3.1 Analyse der Erfüllung der Herausforderungen durch die klassische modellgetriebene Entwicklung

Die modellgetriebene Entwicklung sieht Plattformen vor, in welchen wiederverwendbare Teillösungen formalisiert beschrieben und strukturiert abgelegt werden. Hardware-Software-Bausteine, wie sie in der Automatisierungstechnik vorliegen, können mit den bisherigen Konzepten nicht beschrieben und somit nicht als wiederverwendbare Teillösung in der modellgetriebenen Entwicklung genutzt werden. Daher ist eine Erweiterung der klassischen modellgetriebenen Entwicklung nötig, um Teillösungen mit Hardware- und Softwarebestandteilen wiederverwenden zu können und Hardware-Software-Abhängigkeiten zu berücksichtigen.

Die klassische modellgetriebene Entwicklung ermöglicht die Beeinflussung der Lösung durch das fachliche Modell M(P) sowie die Auswahl der Plattformen, die in der Systementwicklung genutzt werden. Die Vielzahl von geforderten Eigenschaften, die ein Automatisierungssystem aufweisen muss, können in der klassischen modellgetriebenen Entwicklung nicht modelliert und somit nicht erfüllt werden. Um diese Eigenschaften im Rahmen einer modellgetriebenen Entwicklung erfüllen zu können, ist eine Erweiterung erforderlich, um weitere Beeinflussungsmöglichkeiten der Lösung zu schaffen.

Existierende Realisierungen von Transformationen können in der Automatiserungstechnik aufgrund der Vielzahl an Modellierungssprachen und der Hardware-Software-Abhängigkeiten nicht genutzt werden. Um den Aufwand bei der Infrastrukturentwicklung zu reduzieren, sind die generischen Konzepte zur Definition von Transformationsvorschriften weiter zu konkretisieren, indem Eigenschaften der automatisierungstechnischen Teillösungen in den Metamodellen zur Definition der Plattformen berücksichtigt werden. Dies wird in Anlehnung an CAEX [IEC62424] durch ein allgemeines Metamodell für automatisierungstechnische Teillösungen in den Plattformen ermöglicht.

Die Umsetzung dieser Ansätze in ein Konzept zur Hardware-Software-Integration wird im folgenden Kapitel detaillierter beschrieben.

3.2 Konzept der Integrierten Plattform zur Hardware-Software-Integration

Mehrfach verwendbare Teillösungen bestehen in der Automatisierungstechnik aus Hardware- und Softwarebestandteilen. Auf Modellebene bedeutet dies, dass die Verwendung einer Teillösung sowohl Auswirkungen auf die Hardwaremodelle (z. B. den Schaltplan) als auch auf Softwaremodelle wie z. B. den Quellcode hat. Eine Teillösung hat folglich Repräsentationen in unterschiedlichen Modellen. Um diese verschiedenen, zu einer Teillösung gehörenden Repräsentationen zusammenzufassen, wurde das Konzept der Plattformen um Sichten erweitert. Die Plattform für die modellgetriebene Entwicklung in der Automatisierungstechnik besteht somit nicht allein aus Softwarebausteinen sondern aus automatisierungstechnischen Teillösungen. Eine automatisierungstechnischen Teillösung entsteht durch die Kapselung der Repräsentationen der Teillösung für die Hardware- und Softwaremodelle und einer allgemeinen Beschreibung der Teillösung. Die Teillösung zur Beschreibung eines Drehzahlsensors verfügt z. B. über eine Repräsentation für den Stromlaufplan, für das Simulationsmodell und einen Softwaretreiber.

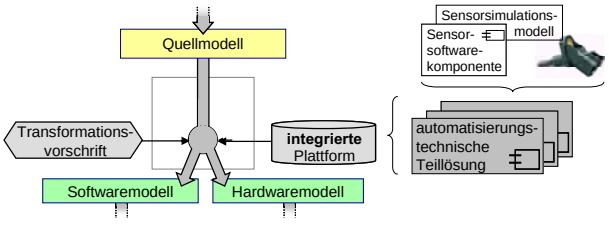


Abbildung 2: Transformation zur Generierung von Hardware- und Softwaremodell mit integrierter Plattform

Alle automatisierungstechnischen Teillösungen werden in einer sogenannten integrierten Plattform strukturiert zusammengefasst. Wird bei der Durchführung einer Transformation durch eine Transformationsregel eine Teillösung aus der Plattform ausgewählt, wird die zugehörige Repräsentation der Teillösung in allen von der Transformation erzeugten Modellen instanziiert. Abbildung 2 verdeutlicht den Ablauf einer Transformation, bei welcher mehrere Zielmodelle aus einem Quellmodell generiert werden.

Das Wissen, welche Teillösung bei der Durchführung einer Transformation ausgewählt werden muss, wie sie zu verknüpfen und zu konfigurieren ist, wird in der Transformationsvorschrift gekapselt. Die Auswahl, Verknüpfung und Konfiguration ist abhängig von einer Vielzahl von Parametern. Um zu vermeiden, dass für jede Kombination an Parameterwerten eine neue Transformationsvorschrift erstellt werden muss, werden die Eigenschaften der Teillösungen formalisiert in der Beschreibung der Teillösung hinterlegt. Transformationsregeln können nun die richtige Teillösung anhand beliebiger, benötigter Eigenschaften selektieren. Die Beschreibung jeder Teillösung folgt dem in Abbildung 3 dargestellten Metamodell. Jede Teillösung verfügt über eine Menge von Eigenschaften (*Feature*), die entweder fest (*Property*) oder optional (*Option*) sind oder über einen Parameter (*Parameter*) konfiguriert werden können.

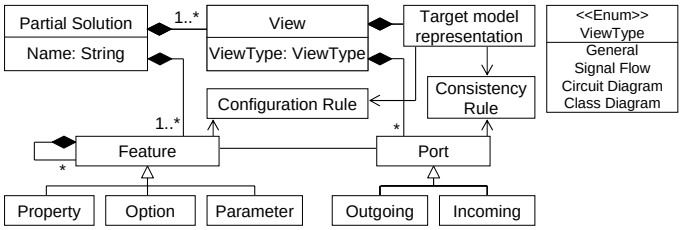


Abbildung 3: Metamodell der Teillösungen der Plattformen für die Automatisierungstechnik

Wie bereits erläutert, stellen Teillösungen der Plattform automatisierungstechnische Teillösungen dar. Die Repräsentationen der Teillösungen in den verschiedenen Zielmodellen werden in der Teillösung gekapselt (*target model representation*). Jede Repräsentation ist dabei einer spezifischen Sicht (*View*) zugeordnet. Über die Anschlüsse (*Port*) wird die Teillösung mit anderen Teillösungen verknüpft.

Da interne Abhängigkeiten zwischen der Hardware und der Software schon bei der Entwicklung der Teillösung bekannt sind, können diese in die Beschreibung der Teillösung als *Configuration Rules* und *Consistency Rules* integriert werden. Diese Regeln werden bei der Durchführung der Transformation ausgewertet. Somit können Anpassungen der Teillösung automatisch veranlasst werden.

Plattformen und Transformationsvorschriften werden a priori entwickelt und dann bei der Entwicklung vieler Systeme wiederverwendet. Dies ermöglicht eine Effizienzsteigerung in der Entwicklung und eine Verkürzung der Entwicklungszeit innerhalb einzelner Projekte. Nachteilig erweisen sich der hohe Initialaufwand zur vollständigen, projektunabhängigen Entwicklung der Transformationsvorschriften und Plattformen und die Inflexibilität bezüglich der Berücksichtigung unterschiedlicher nichtfunktionaler Anforderungen. Eine Lösung dieser Problematik wird im folgenden Abschnitt vorgestellt.

4.3 Konzept der Anpassbarkeit der Transformationen

Um die oben erwähnten verschiedenartigen Anforderungen bei der Entwicklung eines Automatisierungssystems spezifizieren und erfüllen zu können, wurde das Konzept der Transformationen der klasischen modellgetriebenen Entwicklung um zwei Beeinflussungsmöglichkeiten der Lösung erweitert. In Analogie zu Frameworks werden diese Beeinflussungsmöglichkeiten im Folgenden Hot Spots der Transformation genannt.

Durch eine Anpassung der Transformationsvorschriften können globale Eigenschaften eines Systems, d. h. Eigenschaften, die sich aus dem Zusammenwirken einzelner Systemelemente ergeben, berücksichtigt werden. Dies geschieht wie in Abbildung 4 mit (1) gekennzeichnet durch eine Instanziierung und Ergänzung der generischen a priori entwickelten Transformationsvorschrift. Somit ist es möglich, bei der werkzeuggestützten Durchführung der Transformation verschiedene Architekturvarianten hinsichtlich der Erfüllung der globalen Eigenschaften zu analysieren und die Geeignetste auszuwählen. Um beispielsweise eine zuverlässige Temperaturerfassung zu realisieren, muss bei der Transformation zwischen einer Lösung mit einem hoch zuverlässigen Sensor und einer redundanten Messung mit Standardsensoren entschieden werden.

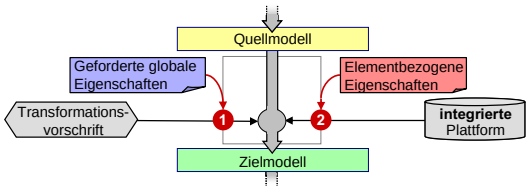


Abbildung 4: Durch Hot Spots anpassbare Transformation in der Automatisierungstechnik

Projektspezifisch geforderte Eigenschaften einzelner Systemelemente wie beispielsweise der Zulieferer einzelner Teillösungen werden durch Selektion und Ausschluss einzelner Teillösungen aus der alle Teillösungen umfassenden Plattform spezifiziert ((2) in Abbildung 4). Nach der Anpassung der Plattform liegen folglich alle Teillösungen vor, die im konkreten Entwicklungsprojekt eingesetzt werden können.

Die Hot Spots erlauben es ferner, die Transformationsvorschrift oder die Plattform innerhalb eines Entwicklungsprojekts zu erweitern. Somit kann die modellgetriebene Entwicklung iterativ eingeführt werden, da in jedem Entwicklungsprojekt die zur Realisierung fehlenden Transformationsregeln und Teillösungen am jeweiligen Hot Spot ergänzt werden können.

Das einheitliche Metamodell der Teillösungen der Plattformen ermöglicht einen gleichartigen Aufbau der Transformationsvorschriften in verschiedenen Domänen der Automatisierungstechnik. Eine Transformationsvorschrift besteht aus vier Typen an Transformationsregeln:

- Strukturaufbauende Regeln kapseln das Wissen zum Aufbau der Struktur des Zielmodells, um das im Quellmodell beschriebene Problem zu lösen. Sie extrahieren relevante Eigenschaften aus dem Quellmodell und erzeugen eine mögliche Struktur des Zielmodells, die durch Teillösungen aus der Plattform detailliert werden muss.

- Selektierende Transformationsregeln dienen zur Selektion der richtigen Teillösung aus der Plattform. Sie beschreiben das Wissen über die relevanten Eigenschaften, die eine Teillösung haben muss, um an einer bestimmten Stelle innerhalb der Struktur des Zielmodells eingesetzt werden zu können. Durch selektierende Transformationsregeln wird zunächst eine komplette Anforderungsspezifikation für die benötigten Teillösungen erstellt. Mithilfe dieser Anforderungsspezifikation wird schließlich die passende Teillösung aus der Plattform ausgewählt.

- Konfigurierende Transformationsregeln sorgen für eine konsistente Konfiguration der selektierten Teillösungen in den einzelnen Zielmodellen.

- Übersetzende Transformationsregeln dienen zur direkten Übersetzung von Informationen des Quellmodells in das Zielmodell ohne Nutzung der Plattform.

Bei der Durchführung einer Transformation werden zunächst die strukturaufbauenden, dann die selektierenden vor den konfigurierenden und übersetzenden Regeln angewandt.

4. Einsatz der modellgetriebenen Entwicklung in einer spezifischen Domäne der Automatisierungstechnik

4.1 Infrastrukturentwicklung

In der Infrastrukturentwicklung wird der Entwicklungsprozess inklusive der eingesetzten Modellierungssprachen definiert. Aus technischer Sicht bedeutet dies, dass die nötigen Transformationen identifiziert und durch die Bereitstellung von Plattform und Transformationsvorschrift realisiert werden. Das fachliche Modell M(P) soll das Problem beschreiben. Aus ihm müssen durch mehrere sequenzielle Transformationen die Modelle der Lösung M(L) generiert werden können. Das Ziel des Automatisierungssystems ist die Führung und Überwachung des technischen Prozesses. Das fachliche Modell in einem durchgängigen modellgetriebenen Entwicklungsprozess ist in der Automatisierungstechnik folglich ein Modell des technischen Prozesses. Zur formalisierten Modellierung des technischen Prozesses werden domänenspezifische Sprachen eingesetzt. Das Konzept zur Definition der domänenspezifischen Sprache zur Modellierung der technischen Prozesse einer Domäne wurde in [Ma08] beschrieben. Die Modelle der Lösung bestehen aus dem Schaltplan des Automatisierungssystems, dem Quellcode der Automatisierungssoftware und Konfigurationsdaten.

Die Anzahl der Transformationen vom fachlichen Modell M(P) zu den Modellen der Lösung M(L) ist durch das Konzept nicht festgelegt. Sie richtet sich nach der nötigen Variabilität der Lösungen innerhalb der Domäne. Denn in jeder Transformation kann die Lösung durch die eingeführten Hot Spots beeinflusst werden. Andererseits resultiert aus jeder Transformation ein manueller Aufwand bei der Entwicklung, da Plattform und Transformationsvorschrift bereitgestellt und bei der Entwicklung eines Automatisierungssystems angepasst werden müssen. Daher ist die Anzahl der Transformationen zu minimieren, ohne die nötige Variabilität der Lösungen zu limitieren.

Zur Entwicklung des Automatisierungssystems werden minimal zwei sequenzielle Transformationen benötigt, da erstens die Funktionalität und Struktur des Automatisierungssystems maßgeblich von der Realisierung des technischen Systems abhängig ist und zweitens die Funktionalität der Automatisierungssoftware von den eingesetzten Sensoren und Aktoren abhängt. Im Rahmen der ersten Transformation können Anforderungen, die sich aus der Realisierung des technischen Systems ergeben durch eine Anpassung der Transformationsvorschrift und der Plattform berücksichtigt werden. In der zweiten Transformation werden Sensoren und Aktoren ausgewählt und die detaillierten Modelle der Lösung erzeugt.

4.2 Modellgetriebene Entwicklung in der Domäne Waschautomat

Aufgabe eines Waschautomaten ist aus Sicht der Automatisierungstechnik die Führung des Waschprozesses innerhalb der Maschine. Der Waschprozess stellt somit das Problem dar, das im fachlichen Modell M(P) beschrieben wird. In diesem Modell wird der zu realisierende Waschprozess in der Technical Process Modelling Language – Washing Process, einer am Institut für Automatisierungs- und Softwaretechnik entwickelten domänenspezifischen Sprache zur Modellierung von Waschprozessen, modelliert.

Aus dem Modell des Waschprozesses können durch zwei sequenzielle Transformationen der Quellcode und der Schaltplan des Automatisierungssystems des Waschautomaten generiert werden. Mit der ersten Transformation wird aus dem Modell des technischen Prozesses das Systemfunktionsmodell erzeugt. Hierbei wird die Funktionsaufteilung zwischen technischem System und Automatisierungssystem festgelegt. Ein Beispiel ist die Türverriegelung des Waschautomaten. Diese kann rein mechanisch durch das technische System oder durch das Automatisierungssystem über einen elektromechanischen Verriegelungsmechanismus realisiert werden. Das Systemfunktionsmodell ist ein Matlab/Simulink/Stateflow-Modell, in welchem das Verhalten des Automatisierungssystems und des technischen Systems simuliert werden kann. Die in diesem Modell spezifizierten Funktionen des Automatisierungssystems werden in der zweiten Transformation realisiert. Hierbei werden Quellcode und Schaltplan des Automatisierungssystems generiert.

5. Werkzeugunterstützung

Zur Demonstration des Ansatzes wird am Institut für Automatisierungs- und Softwaretechnik eine auf Eclipse EMF [EMF08] und openArchitectureWare [oAW08] basierende Entwicklungsumgebung erstellt. Sie unterstützt die Definition von domänenspezifischen Sprachen zur Modellierung der technischen Prozesse für verschiedene Domänen. Realisiert wurde ein Editor für die Modellierung von Waschprozessen. Die Transformationen wurden in openArchitectureWare realisiert. Zur Definition der Transformationsvorschrift wird die von openArchitectureWare bereitgestellte Sprache Xtend genutzt. Das Metamodell der Teillösungen der Plattform wurde als ecore-Modell spezifiziert. Die Beschreibung der Teillösungen wird im XML-Format abgelegt und bei der Durchführung der Transformation geladen. Somit können neue Teillösungen einfach ergänzt werden.

6. Zusammenfassung

Bedingt durch kürzere Entwicklungszeiten und den Kostendruck im weltweiten Wettbewerb entsteht ein großer Bedarf, die Entwicklung von Automatisierungssystemen effizient durchzuführen. In der modellgetriebenen Entwicklung werden zielgerichtete Abstraktion eingesetzt, um eine Konzentration auf die relevanten Aspekte zu ermöglichen, und Transformationen genutzt, um aus den abstrakten Modellen des Problems detaillierte Modelle der Lösung automatisiert zu generieren.

Integrierte Plattformen, Hot Spots in der Transformation und eine metamodellbasierte Definition der Teillösungen in der Plattform erlauben im vorgestellten Konzept eine automatisierte Berücksichtigung von Hardware-Software-Abhängigkeiten, die Erfüllung verschiedenartiger Anforderungen an Automatisierungssysteme und den Einsatz unterschiedlicher Modellierungssprachen. Dies ermöglicht eine modellgetriebene Entwicklung von Automatisierungssystemen und führt zu einer besseren Beherrschbarkeit der Komplexität, einer Steigerung der Effizienz sowie eine Verkürzung der Entwicklungszeit innerhalb einzelner Automatisierungsprojekte.

Literaturverzeichnis

[CzHe03] K. Czarnecki, U. Helsen: Classification of Model Transformation Approaches. OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[EMF08] Eclipse Modeling Framework Projekt: www.eclipse.org/modeling/emf, 2008.

[IEC62424] IEC: Representation of process control engineering - Requests in P&I diagrams and data exchange between P&ID tools and PCE-CAE tools. IEC 62424, 2008.

[Ma08] M. Maurmaier: Leveraging Model-driven Development for Automation Systems Development, IEEE ETFA 2008, Hamburg, 2008.

[OMG03] Object Management Group: MDA Guide Version 1.0.1, 2003.

[oAW08] Homepage des openArchitectureWare Projekts: www.openarchitectureware.org, 2008.

[QVT08] Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation, Version 1.0, 2008.

[Sch08] B. Schenk, M. Schlereth: Model Driven Development applied to Automation Engineering, Automation 2008, pp. 156, 2008.

[SmBr93] G. Smith, G. Brown: Conceptual Foundations of Design Problem Solving, IEEE Transactions on Systems, Man, and Cybernetics; Vol. 23, No. 5, 1993.

[Som07] J. Sommerville: Software Engineering, 8. Ausgabe, Addison Wesley, 2007.

[ZVEI06] ZVEI Fachverband Automation: Integrierte Technologie-Roadmap Automation 2015+, Essen, 2006.

Semantic-Preserving Test Model Transformations for Interchangeable Coverage Criteria

Stephan Weißleder
Humboldt-Universität zu Berlin, Rudower Chaussee 25, D-12489 Berlin
weissled@informatik.hu-berlin.de

Abstract: Testing cannot be complete. Heuristic means like coverage criteria are applied to measure the quality of tests. In model-based testing, it is common to apply coverage criteria to test models. Beyond test quality measurement, coverage criteria are also used to steer test generation. However, model-based test generators are often restricted to satisfy a limited set of coverage criteria. In this paper, we present test model transformations as an alternative to the satisfaction of unsupported coverage criteria. We show several transformations for different coverage criteria. Thus, model transformations can be understood as a means to make coverage criteria interchangeable. The foundation of this work are experiences from an industrial cooperation.

1 Introduction

Testing is a very important system validation technique. Test suites are executed on a system under test (SUT). Each test suite is a set of test cases. In functional testing, each test case is a sequence of input stimuli and expected system behaviour. With the success of models in system development, models also gained importance for testing. In model-based testing, test models are used as test specifications. Several modeling languages have already been used to create test models (e.g., B, Z, UML, OCL). We concentrate on state machines of the Unified Modeling Language (UML) 2.1 [Obj07].

For many non-trivial SUTs, it is infeasible to prove their correctness, nor can testing guarantee the absence of faults. Due to this absence of absolute means of quality measurement, heuristic means are used. Coverage criteria are such means. Their relation to the test suite's fault detection ability is still a relevant research topic [PPW$^+$]. Besides measuring quality, coverage criteria are also used to steer automatic test generation. They are, however, independent of the test model and have to be transformed into test model-specific test goals for further processing. For instance, the coverage criterion All-Transitions is transformed into test goals that reference one transition of the state machine, each. This notion of test goals is applied in many commercial tools, e.g. Leirios LTD [Lei]. The remaining task is to create test paths that satisfy the test goals (e.g., by traversing the referenced transitions).

In conventional testing, coverage criteria are often directly applied to the source code of the SUT [AO08, page 52]. The application of coverage criteria to test models instead introduces a gap caused by the higher abstraction level of the test model compared to the SUT. A UML state machine is a behavioural abstraction of the SUT. The transformation

of a coverage criterion into a set of test goals, however, is influenced by the structure of the test model – independent of the described behaviour. In this paper, we focus on transforming the structure of state machines while preserving their described behaviour.

Such transformations of test models bear several benefits. For instance, existing test generators are often able to satisfy only a restricted set of coverage criteria. Test model transformations are presented as a means to satisfy unsupported coverage criteria. Furthermore, test model transformations are a valuable alternative to the development of new or combined coverage criteria: Instead of implementing an unsupported or combined coverage criterion, a supported coverage criterion can be satisfied on a transformed test model.

The paper is structured as follows. The following section contains the related work. In Section 3, our experiences from an industrial cooperation are described, which are the foundation of this paper. Section 4 contains a description of the mutual dependency of test models and coverage criteria together with several test model transformations to support test generators. The final section contains conclusion, discussion, and our future work.

2 Related Work

Several books provide surveys of conventional testing [Mye79, Bin99, AO08] and model-based testing [UL06]. Modeling languages like the UML [Obj07] have been used to create test models. For example, Abdurazik and Offutt provide an approach for automatic test generation from state machines [OA99]. More work about testing with UML can be found, e.g. in [SHS03, BLC05, WS07]. In this paper, UML state machines are used as test models.

Coverage criteria are popular to measure the quality of test suites. Several types of coverage criteria have been defined and investigated (e.g., focused on data flow, control flow, or boundary value analysis). For instance, data flow-oriented coverage criteria are considered by Weyuker [Wey93]. Kosmatov et al. define boundary-based coverage criteria [KLPU04]. Coverage criteria can be related by subsumption and there are many approaches to define new coverage criteria that subsume existing ones without causing exponential effort. Other approaches aim at the combination of coverage criteria, as described in [WS08] or [FSW08]. In contrast to the cited work, test model transformations are a more general means to support the satisfaction of coverage criteria.

The impact of the test model's structure on the test suite quality has only been investigated for a few scenarios. For instance, Rajan et al. [RWH08] examine the impact of the model's and the program's structure on the satisfaction of Modified Condition / Decision Coverage (MC/DC) [CM94]. Ranville [Ran03] proposes a way to satisfy MC/DC by traversing all transitions on a changed test model. Friske and Schlingloff [FS07] transform the test model so that the satisfaction of MC/DC on the transformed test model has the same effect as the satisfaction of All-Transition-Pairs [UL06, page 118] on the original test model. Compared to the cited work, this paper presents a set of test model transformations that are used to make a set of different coverage criteria interchangeable.

For test generation in model-based testing, coverage criteria are applied to test models. The test suite quality, however, is measured at the SUT, e.g. with mutation analysis. In

mutation analysis, mutation operators inject faults in a correct SUT. The more mutated SUTs (mutants) are detected by the test suite, the higher is its fault detection ability. Many mutation operators have been defined [OL94, OLR+96, BOY00]. Corresponding case studies [ABL05, Par05, ABLN06, NAM08] substantiate that mutation analysis is a good predictor for the test suite's fault detection ability of real faults. In this paper, we apply mutation analysis.

3 Industrial Cooperation

Our motivation to investigate test model transformations results from experiences of an industrial cooperation with a German rail engineering company. In this cooperation, we had to generate test suites from a UML state machine. This state machine contained two parallel regions and composite states with a hierarchy depth of 4. It comprised about 35 states and 70 transitions, and described the behaviour of a train control.

Since the real SUT was not provided, we manually created correct implementations of the test model, applied mutation analysis to them, and measured the fault detection ability of the generated test suites. The first evaluation results were unsatisfying. We investigated the reasons and came up with several solutions, including the transformation of the test model. We implemented some test model transformations in our prototype ParTeG [Wei]. Since the transformations are executed in a preprocessing step, they can be externalised and used for any other model-based test generation tool.

Table 1 shows the results of mutation analysis for test suites based on the original and the transformed test model (TC – transition coverage; MC/DC – modified condition / decision coverage; MCC – multiple condition coverage). Note that the number of killed mutants significantly increased after the transformation and that the test suite size increased only moderately ($< +30\%$). Usually, the test suite size increases exponentially with increasing (mutation) coverage (cp. [ABLN06]). The remaining mutants were killed by applying boundary coverage criteria [KLPU04] to the input partitions of the state machine paths.

The impact of the used test model transformations is not restricted to our implementations. Our industrial partner reported that the test suites from the transformed test model detected more faults than the test suites derived from the original test model.

Coverage Criterion	Original Test Model		Transformed Test Model	
	Test Suite Size	Killed Mutants	Test Suite Size	Killed Mutants
TC	117	610 / 872	148	661 / 872
masking MC/DC	197	790 / 872	229	843 / 872
MCC	257	810 / 872	288	863 / 872

Table 1: Results of mutation analysis in the industrial cooperation.

4 On the Importance of Test Models in Model-Based Testing

The results of the cooperation showed the importance of test model transformations: They can be used as a means to increase the effect of a satisfied coverage criterion. Since many industrial model-based test case generators support a limited set of coverage criteria [BSV08], such transformations are of high value. For instance, Leirios Test Designer [Lei] supports only transition coverage on flattened state machines.

This section contains descriptions of several scenarios in which test model transformations are useful to satisfy or to combine coverage criteria. It starts with a rationale for the reduced effect of coverage criteria.

4.1 Quality Measurement at Different Levels of Abstraction

In model-based testing, coverage criteria are applied to test models. Furthermore, the SUT is hidden. Thus, coverage criteria cannot be applied to the SUT. Since coverage criteria are test-model-independent, they are transformed to test-model-specific test goals. We use state machines as test models. The test goals depend on the structure of the state machine.

Mutation analysis is often based on mutation operators (cp. [AO08, page 182]). These mutation operators change small details (e.g., they exchange relational or logical operators) and, thus, depend on the structure of the correct implementation of the SUT.

Both means of quality measurement depend on the structure of the state machine respectively the SUT. State machine and SUT, however, are only related by behavioural abstraction. Their structures are independent of each other. From this independency, it follows that the test goals and the mutation operators are independent of each other. Consequently, the selected coverage criterion and the result of mutation analysis are independent. Since coverage criteria are understood as an important means to influence the test suite quality, this conclusion means an important issue. Figure 1 depicts this scenario.

In the following, we present several test model transformations. These transformations preserve the described behaviour of the test model but change its structure. Thus, the test model is still a behavioural description of the SUT. As we will show, however, the impact of the applied coverage criterion will be changed.

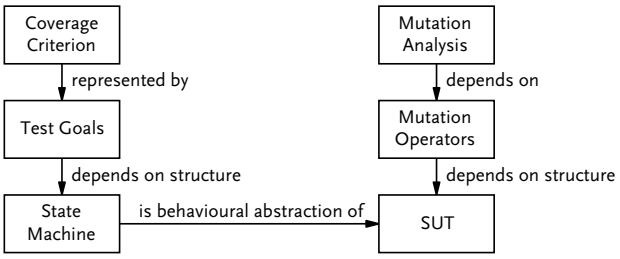


Figure 1: Structural independency of coverage criteria and mutation analysis.

4.2 Transform Test Models to Simulate the Satisfaction of Coverage Criteria

This section contains the description of several structural test model transformations. The satisfaction of supported coverage criteria on transformed test models can be used to satisfy unsupported coverage criteria on the original test model.

Definition 1 (Simulated Satisfaction) *Assume a test generator supports the satisfaction of a coverage criterion CC_S on the test model level but does not support the satisfaction of a coverage criterion CC_N. If there is a model transformation so that each test suite that satisfies CC_S on the transformed test model also satisfies CC_N on the original test model, then the satisfaction of CC_S can be used to* simulate the satisfaction *of CC_N.*

Utting and Legeard [UL06] subdivide structural model-based coverage criteria into several kinds, e.g. control-flow-oriented, data-flow-oriented, and transition-based. Friske and Schlingloff [FS07] add variables to the test model and simulate the satisfaction of All-Transition-Pairs on the original test model by satisfying MC/DC on the transformed test model with Rhapsody ATG [Tel]. Their work provides an example for the simulated satisfaction of a transition-based coverage criterion by satisfying a control-flow-oriented coverage criterion. In the following, we list further examples of simulated satisfaction for the remaining combinations of structural model-based coverage criteria.

4.2.1 Simulated Satisfaction of Transition-Based Coverage Criteria by the Satisfaction of Data-Flow-Oriented Coverage Criteria

The transition-based coverage criterion All-Transition-Pairs is satisfied iff all transition sequences up to length 2 are traversed. The data-flow-oriented coverage criterion All-Uses [UL06, page 115] is satisfied iff all def-use-pairs of variables are tested. This section contains a description of how to simulate the satisfaction all All-Transition-Pairs by satisfying All-Uses. We assume a test generator that supports the satisfaction of All-Uses.

A model transformation that makes this simulated satisfaction possible consists of the following: For each vertex *v* in the state machine, a new attribute *a* is defined in all of *v*'s incoming transitions and used in all of *v*'s outgoing transitions. For traversing the same transition twice, the definition of *a* is added behind its use. Each *a* is removed again after test suite generation. Figure 2 depicts such a model transformation for the state *S3*. If all the new def-use-pairs are tested and All-Uses is satisfied, then also all transition pairs are traversed and All-Transition-Pairs is satisfied. We prove the contraposition of this statement: If All-Transition-Pairs is unsatisfied, then there is at least one pair of transitions (t_1, t_2) that is not traversed in sequence. Reasons for that can be that t_1 or t_2 are not traversed at all or that additional transitions t_a are traversed between t_1 and t_2 with each t_a being unequal to t_1 and t_2. In the first case, the newly inserted def-use-pair corresponding to t_1 and t_2 is not tested and All-Uses is not satisfied. In the second case, t_2 has to be traversed after t_a. For that, t_2's source state must be reached and an incoming transition t_i of t_2's source state must be traversed. Since all transitions that lead to that state redefine *a* and t_i is unequal to t_1, the original def-use-pair is not tested and All-Uses is not satisfied.

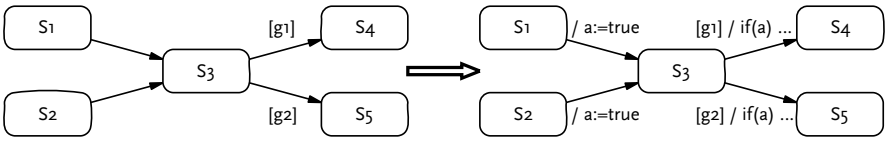


Figure 2: Test model transformation for simulated satisfaction of All-Transition-Pairs.

4.2.2 Simulated Satisfaction of Control-Flow-Oriented Coverage Criteria by the Satisfaction of Transition-Based Coverage Criteria

Control-flow-oriented coverage criteria, e.g. MC/DC [CM94], are satisfied if a certain selection of value assignments for each guard is tested. In this section, a test generator that is able to satisfy All-Transitions should be used to simulate the satisfaction of MC/DC. Figure 3 depicts one possible corresponding test model transformation: For each value assignment *va* of a transition guard necessary to satisfy MC/DC, a new transition is created with a logical conjunction that represents *va* as transition guard. The original transition is removed if at least one new transition is created. The traversal of each transition on the adapted test model implies the test of each necessary guard value assignment *va*. Consequently, the satisfaction of All-Transitions on the adapted test model simulates the satisfaction of MC/DC on the original test model.

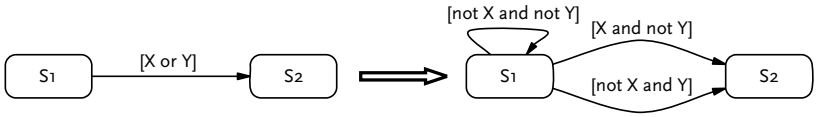


Figure 3: Test model transformation for simulated satisfaction of MC/DC.

4.2.3 Simulated Satisfaction of Control-Flow-Oriented Coverage Criteria by the Satisfaction of Data-Flow-Oriented Coverage Criteria

This section contains a description of how to simulate the satisfaction of the control-flow-oriented coverage criterion MC/DC by satisfying the data-flow-oriented coverage criterion All-Uses. This simulated satisfaction is achieved by combining the model transformations of Section 4.2.2 and 4.2.1: The transformation of Section 4.2.2 is used to simulate the satisfaction of by MC/DC by satisfying All-Transitions. All-Transition-Pairs subsumes All-Transitions and the transformation in Section 4.2.1 is used to simulate the satisfaction of All-Transition-Pairs by satisfying All-Uses. Consequently, the satisfaction of All-Uses after both model transformations simulates the satisfaction of All-Transition-Pairs and All-Transitions after one test model transformation, and also the satisfaction of MC/DC on the original test model.

4.2.4 Simulated Satisfaction of Data-Flow-Oriented Coverage Criteria by the Satisfaction of Control-Flow-Oriented Coverage Criteria

In this section, the simulated satisfaction of the data-flow-oriented coverage criterion All-Uses by satisfying the control-flow-oriented coverage criterion MC/DC is shown. Figure 4 depicts a possible test model transformation: For each def-use-pair (d_v, u_v) of a variable *v*, a new boolean attribute *b* is defined. At d_v, *b* is set to true and at u_v, the guard condition of the corresponding transition is combined with *[b = true]* by disjunction. At the initial node and at all other definitions of *v*, the value of *b* is set to false. To satisfy MC/DC, *b* has to be true in at least one value assignment of the transformed guard of u_v. For that, *v* has to be defined with at d_v and not redefined before being used u_v. If this holds for all *v*, then All-Uses is satisfied. This transformation is similar to the one presented in [FS07]. This time, however, def-use-pairs are instrumented instead of transition pairs.

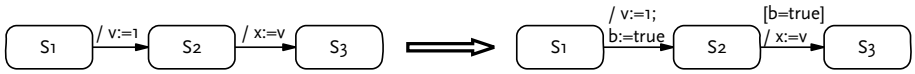


Figure 4: Test model transformation for simulated satisfaction of All-Uses.

4.2.5 Simulated Satisfaction of Data-Flow-Oriented Coverage Criteria by the Satisfaction of Transition-Based Coverage Criteria

To simulate the satisfaction of the data-flow-oriented coverage criterion All-Uses by the transition-based coverage criterion All-Transitions, we combine two simulated satisfaction relations just like in Section 4.2.3 and compose the test model transformations shown in Section 4.2.2 and Section 4.2.4: The simulated satisfaction of All-Uses by satisfying MC/DC (Section 4.2.4) is applied on the simulated satisfaction of MC/DC by satisfying All-Transitions (Section 4.2.2). The result is that the satisfaction of All-Uses is simulated by satisfying All-Transitions with the combination of both model transformations.

4.3 Combination of Coverage Criteria

Besides the simulated satisfaction of coverage criteria, test model transformations can also be used to combine different coverage criteria. There are several ways to combine coverage criteria [FSW08]. The first and very simple one is achieved by combining test suites that satisfy different coverage criteria. With the approach of test model transformation, this effect can be reached easily by applying two times a different test model transformation on the original test model, generating both test suites, and combining them, afterwards. For instance, with a test generator that is only capable of satisfying All-Transitions (e.g., Leirios LTD [Lei]), a test suite can be generated that satisfies MC/DC (Section 4.2.2) as well as All-Uses (Section 4.2.5). This is likely to result in redundant test cases and there is much room for optimization (cp. [FSW08]), e.g. by monitoring test goal satisfaction.

There are, however, more complicated combinations of coverage criteria. For instance, the goal may not be to independently simulate the satisfaction of two coverage criteria like All-Transition-Pairs and MC/DC but to combine them by satisfying MC/DC for each pair of adjacent transitions. The advantage of this combination can be clarified with the state machine visualized on the left in Figure 5. The sole satisfaction of MC/DC can miss a test case in which *[X and (not Y)]* (which is necessary to satisfy MC/DC on *[X and Y]*) is included in a path that visits the state *S1*. It could have been included in a path that visits the state *S2*, instead. The sole satisfaction of All-Transition-Pairs can miss a test case in which *[X and (not Y)]* is satisfied at all. The reason for this is that the else-guard can also be satisfied by, e.g. *[(not X) and Y]*. Both issues are solved if MC/DC is satisfied for each pair of adjacent transitions. An alternative to simulate the satisfaction of All-Transition-Pairs like in Section 4.2.1 is to duplicate each transition pair's intermediate state *S3* and its outgoing transitions for each of *S3*'s incoming transitions (see Figure 5). The satisfaction of MC/DC on this transformed test model simulates the satisfaction of MC/DC for each pair of adjacent transitions on the original test model. Furthermore, this satisfaction of MC/DC can be simulated by satisfying All-Transitions as proposed in Section 4.2.2.

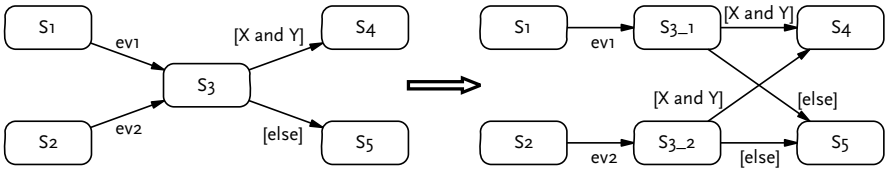


Figure 5: Test model transformation for combining All-Transition-Pairs and MC/DC.

5 Conclusion, Discussion, and Future Work

In this paper, we presented several model transformations as a means to interchange coverage criteria and to support model-based test generators. Furthermore, we sketched how to use test model transformations to combine coverage criteria. We also reported of an industrial cooperation which provided the motivation for this paper. The results of the cooperation substantiated the importance of test model transformations in realistic scenarios.

The presented test model transformations are a serious alternative to coverage criteria development. There are, however, some points left to discuss. For instance, test model transformations are no solution to the general issue that coverage criteria are just heuristic means of quality measurement. Since proofs for a system's correctness are rare, however, the uncertainty related to these heuristics is widely accepted. Furthermore, test model transformations can considerably increase the test model size. As long as the simulated satisfied coverage criterion has no exponential effort, we did not detect an exponential effort for the transformed test model so far. Beyond, manual test model transformations can be a tedious and error-prone. Since all presented transformations can be automated in a tool, however, they are hidden from the user and mean no additional manual effort. Finally,

we only showed the general feasibility of combining and simulating different kinds of coverage criteria by a few examples such as All-Uses, All-Transitions, or MC/DC. Indeed, we do not claim the generality of simulated satisfaction for all combinations of different coverage criteria. The used coverage criteria, however, are perceived as important. The shown transformations provide a good foundation for further investigations.

The presented test model transformations are a valuable alternative to coverage criteria development and we plan to elaborate our presented contribution. This includes the definition of an explicit relation between coverage criteria that are applied to different test models. The used coverage criteria are of different type. This brings up the question how different coverage criteria really are with respect to model transformations. Some test model transformations are already implemented in the test generator ParTeG [Wei]. Since the support of existing commercial test generators is an important task, a separate model transformation engine that is independent of test generation is necessary to provide the benefits of our work to the many users of the existing test tools.

Acknowledgements. This work was supported by grants from the DFG (German Research Foundation, research training group METRIK).

References

[ABL05] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05*, pages 402–411, New York, NY, USA, 2005. ACM.

[ABLN06] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Software Engineering, IEEE Transactions on*, 32:608–624, 2006.

[AO08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[Bin99] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[BLC05] L. C. Briand, Y. Labiche, and J. Cui. Automated support for deriving test requirements from UML statecharts. *Software and Systems Modeling*, V4(4):399–423, November 2005.

[BOY00] P. E. Black, V. Okun, and Y. Yesha. Mutation Operators for Specifications. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 81, Washington, DC, USA, 2000. IEEE Computer Society.

[BSV08] C. J. Budnik, R. Subramanyan, and M. Vieira. Peer-to-Peer Comparison of Model-Based Test Tools. In *GI Jahrestagung (1)*, volume 133 of *LNI*, pages 223–226. GI, 2008.

[CM94] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. In *Software Engineering Journal*, 1994.

[FS07] M. Friske and B.-H. Schlingloff. Improving Test Coverage for UML State Machines Using Transition Instrumentation. In Francesca Saglietti and Norbert Oster, editors, *SAFECOMP*, volume 4680 of *Lecture Notes in Computer Science*, pages 301–314. Springer, 2007.

[FSW08] M. Friske, B.-H. Schlingloff, and S. Weißleder. Composition of Model-based Test Coverage Criteria. In *MBEES'08: Model-Based Development of Embedded Systems*, 2008.

[KLPU04] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary Coverage Criteria for Test Generation from Formal Models. In *ISSRE '04*, pages 139–150. IEEE, 2004.

[Lei] Leirios. LTD/UML. http://www.leirios.com.

[Mye79] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, USA, 1979.

[NAM08] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE '08*, pages 351–360, New York, NY, USA, 2008. ACM.

[OA99] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In *UML'99*, pages 416–429, 1999.

[Obj07] Object Management Group. Unified Modeling Language (UML), version 2.1, 2007.

[OL94] A. Jefferson Offutt and Stephen D. Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[OLR+96] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, pages 99–118, 1996.

[Par05] A. Paradkar. Case studies on fault detection effectiveness of model based test generation techniques. In *A-MOST '05*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[PPW+] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401.

[Ran03] S. Ranville. MCDC Test Vectors From Matlab Models – Automatically. In *Embedded Systems Conference*, San Francisco, USA, 2003.

[RWH08] A. Rajan, M. W. Whalen, and M. P. E. Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In *ICSE '08*, pages 161–170, New York, NY, USA, 2008. ACM.

[SHS03] D. Seifert, S. Helke, and T. Santen. Test Case Generation for UML Statecharts. In *PSI03*. Springer-Verlag, 2003.

[Tel] Telelogic. Rhapsody Automated Test Generation. http://www.telelogic.com.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2006.

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

[Wey93] E. J. Weyuker. More Experience with Data Flow Testing. *IEEE Trans. Softw. Eng.*, 19(9):912–919, 1993.

[WS07] S. Weißleder and B.-H. Schlingloff. Deriving Input Partitions from UML Models for Automatic Test Generation. In *MoDELS Workshops*, volume 5002 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2007.

[WS08] S. Weißleder and B.-H. Schlingloff. Quality of Automatically Generated Test Cases based on OCL Expressions. In *ICST*, pages 517–520. IEEE Computer Society, 2008.

Variantenmodellierung und -konfiguration eingebetteter automotive Software mit Simulink

Christian Dziobek[1)], Jens Weiland[2)]

[1)]Mercedes-Benz Cars Entwicklung
Daimler AG, D-71059 Sindelfingen
christian.dziobek@daimler.com

[2)]Hochschule Reutlingen, Fakultät Technik
Alteburgstr. 150, D-72762 Reutlingen
jens.weiland@reutlingen-university.de

Abstract: In eingebetteter automotive Software spiegelt sich die Vielzahl möglicher Funktionsvarianten heutiger Fahrzeuge wider. Besondere Herausforderung liegt hierbei im Handling dieser Variabilität. Der Beitrag beschreibt auf der Basis des Engineerings von Systemfamilien und der Generativen Softwareentwicklung Konzepte zur Modellierung und Konfiguration von Funktionsvarianten in eingebetteter automotive Software mit Simulink.

1 Einleitung

Eingebettete automotive Software ist gekennzeichnet durch ein hohes Maß an Variabilität. Aufgrund einer Vielzahl von Fahrzeugkonfigurationen und Anforderungen infolge unterschiedlicher Hardware, Absatzmärkte und Regularien, gibt es vielschichtige Aspekte dieser Variabilität [SZ03].

Die Entwicklung der eingebetteten Software geschieht zunehmend modellbasiert und durch den Einsatz von Codegeneratoren. Ein häufig verwendetes SW-Werkzeug ist die MATLAB Werkzeugkette von The Mathworks mit den Toolboxen Simulink und Stateflow. Diese graphischen Modellierungssprachen ermöglichen die Spezifikation, Modellierung und Simulation signalfluss- und zustandsorientierter Systeme. Simulink-Modelle sind aus der Verschaltung von Elementarblöcken und Statecharts aufgebaute Signalflussgraphen. Komplexe Teilfunktionen können durch „Subsysteme“ gekapselt werden, die wiederum als Signalflussgraphen modelliert sein können. Die Signalverbindungen repräsentieren die Daten, die während der Simulation eines Modells zwischen den Blöcken ausgetauscht werden. In Simulink stehen zahlreiche Blöcke für verschiedenste Funktionen zur Verfügung; beispielsweise Blöcke für logische Operationen oder das Routing von Signalen. Jeder Block besitzt eine Reihe von Parametern, durch deren Konfiguration sich Darstellung, Eigenschaften und das Verhalten des Blocks anpassen lassen.

Um der Variabilität in eingebetteter automotive Software Rechnung zu tragen, ist für eine wirtschaftliche Betrachtung die Wiederverwendung, z.B. von Spezifikationen/ Modellen, Architekturen, Komponenten oder auch Dokumentation und Tests, essentiell. Die modellbasierte Softwareentwicklung erlaubt mit dem durchgängigen Einsatz von Modellen über alle Entwicklungsphasen und der Abstraktion von einer konkreten Zielplattform bereits einen wichtigen Schritt in Richtung Wiederverwendung. Soll Variabilität darüber hinaus berücksichtigt werden, z.B. in Form variabler Daten oder Funktionen, bedarf es zwingend weitergehender Konzepte. Insbesondere beim Einsatz von Simulink und Stateflow zeigt sich in der Praxis,

- dass auf Grund der heute nur unzureichend vorhandenen Beschreibungsmittel sowohl strukturelle Variabilitäten als auch kompositorische Variabilitäten in Simulink nicht eindeutig und prozesssicher[1] beschrieben werden können. Eine sichere Zuordnung durch den Modellierer oder von nachgeschalteten Codegeneratoren ist somit nicht gegeben,

- dass Variabilitäten und deren Abhängigkeiten in den Modellen verborgen bleiben; eine anwendungsspezifische Verwaltung und Darstellung von Variabilität ist somit nicht möglich; dies ist aber gerade im Rahmen der Weiterentwicklung der Modelle und effizienten Zusammenarbeit zwischen Automobilherstellern und Zulieferern unabdingbar und

- dass auf Grund der fehlenden Konzepte zur Handhabung von Variabilität in Simulink ein Verifizieren gültiger Modellkonfigurationen nicht möglich ist.

Der vorliegende Beitrag beschreibt einen Ansatz, wie man diese Defizite verringern kann. Im Abschnitt 2 werden zunächst die zu Grunde liegenden Konzepte und die daraus abgeleiteten Erfordernisse an eine Variantenmodellierung und -konfiguration mit Simulink vorgestellt. Abschnitt 3 beschreibt anschließend die dafür relevanten Konzepte. Deren Anwendung wird in Abschnitt 4 erläutert. Abschnitt 5 fasst die Ergebnisse zusammen.

2 Ausgangspunkt einer Variabilitätsbetrachtung mit Simulink

Die Variantenmodellierung und -konfiguration eingebetteter Software mit Simulink basiert auf dem Engineering von Systemfamilien und der Generativen Softwareentwicklung. Statt der Entwicklung einzelner Systeme fokussieren Systemfamilien eine Gruppe von Systemen [CN01, CE00, GS04]. Ihre Entwicklung verläuft im Wesentlichen in zwei parallelen Prozessen: Dem Domain Engineering und dem Application Engineering. Ziel des *Domain Engineerings* ist die Schaffung einer Infrastruktur für die Wiederverwendung von Artefakten, wie beispielsweise Anforderungsmodelle, eine für

[1] Prozesssicherheit wird in diesem Kontext als Workflow verstanden, der zu jedem Zeitpunkt der Softwareentwicklung Variabilität verfolgbar und deren Auswirkungen eindeutig nachvollziehbar macht.

alle Mitglieder der Systemfamilie übergreifende Softwarearchitektur und Komponenten, die in den Phasen des Domain Engineerings entwickelt und implementiert werden.

Während des *Application Engineerings* bilden diese Artefakte die Grundlage, auf der konkrete Mitglieder der Systemfamilie realisiert werden. Die Realisierung eines Familienmitglieds kann hierbei von einer manuellen Anpassung und Integration der Softwarearchitektur und Komponenten auf Basis von Produktionsplänen bis hin zu einer automatisierten Produktion über eine Werkzeugkette geschehen.

Die Generative Softwareentwicklung verfolgt das Ziel die individuellen Mitglieder einer Systemfamilie automatisiert zu erzeugen. Wesentliches Konzept ist hier das Generative Domänenmodell (s. Abbildung 1). Dieses trennt Konzepte der Anwendungsdomäne im Problemraum von den Konzepten der Implementierung im Lösungsraum. Es ermöglicht eine getrennte Entwicklung von Domänenkonzepten und wiederverwendbaren Komponenten und somit deren individuelle Modellierung, Implementierung und Evolution. Beide Modelle können sich auf diese Weise unabhängig voneinander entwickeln. Das Konfigurationswissen bildet den Problemraum auf den Lösungsraum ab und stellt die Beziehungen zwischen beiden Modellen explizit heraus.

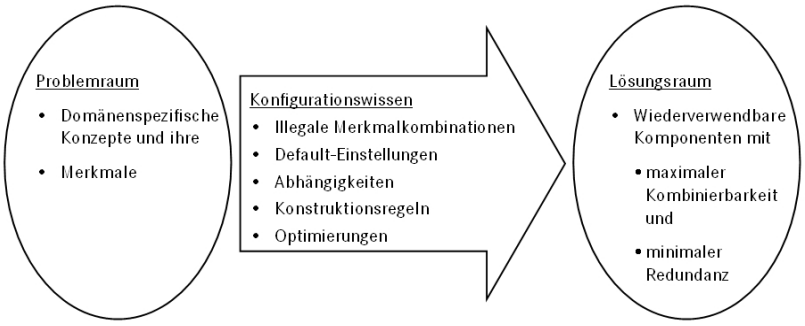


Abbildung 1: Elemente des generativen Domänenmodells (aus [CE00])

Nimmt man das Engineering von Systemfamilien und die Generative Software-entwicklung als Basis für die Variantenmodellierung und -konfiguration mit Simulink, bedarf es dezidierter Konzepte

- das Domain Engineering betreffend für das Management domänenspezifischer Variabilität, die Handhabung dieser Variabilität im Simulink-Modell, sowie das Mapping der beiden Konzepte (s. Abschnitt 3),

- das Application Engineering betreffend für die Spezifikation und automatisierte Konfiguration von gültigen Modellvarianten aus einem variantenreichen Simulink-Modell (s. Abschnitt 4).

3 Entwurf variantenspezifischer Konzepte

3.1 Management domänenspezifischer Variabilität mittels Merkmalmodelle

Für das Management gemeinsamer und variabler Eigenschaften einer Systemfamilie wurde die Merkmalmodellierung verwendet. Merkmalmodelle bilden eine abstrakte Darstellung der Variabilität in einer Systemfamilie [Be03, CE00, LK02]. Im Merkmalmodell werden die gemeinsamen und variablen Eigenschaften einer Systemfamilie in Form von Merkmalen, sowie deren Abhängigkeiten, verwaltet und hierarchisch strukturiert. Merkmale werden unterschieden nach verbindlichen und optionalen Merkmalen, sowie (1..n):m-Gruppenbeziehungen.

Mit Merkmalen können beliebige Informationen assoziiert werden, entweder als Dokumentationsmittel oder auf formaler Ebene, um z.B. automatisierte Konfigurationsprozesse zu bedienen (wie im Folgenden noch gezeigt wird).

3.2 Handling von Variabilität mit Simulink

Ausgangspunkt für die Beschreibung von Variabilität in Simulink-Modellen ist der *Variationspunkt* (s. Abbildung 2). Dieser kapselt die Variabilitätsinformationen, mit denen das Simulink-Modell angereichert werden soll.

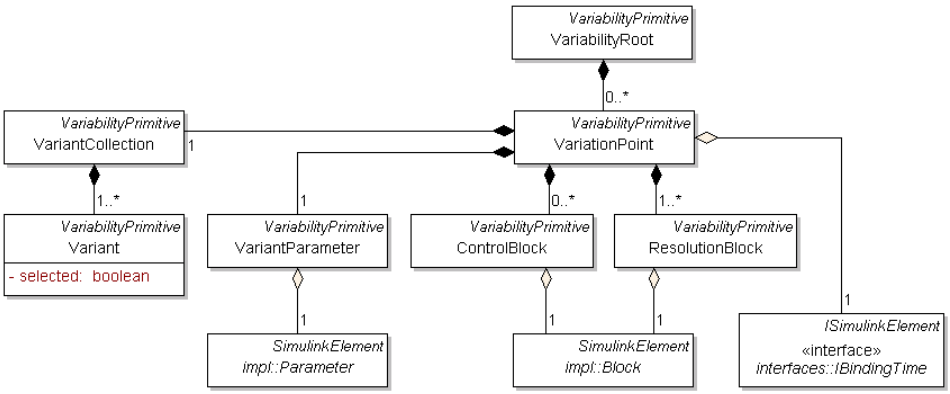


Abbildung 2: Konzeption eines Variationspunktes

Wesentliches Element eines Variationspunktes ist sein eindeutiger *Variantenparameter*. Er stellt die eigentliche „Stellschraube“ im Simulink-Modell dar, die letztendlich konfiguriert wird, um eine Funktionsvariante zu erzeugen. Jeder Variantenparameter besitzt eine *Variantenkollektion*; seine Wertemenge, aus der er konfiguriert werden kann. Die Bindezeit spezifiziert, wann diese Konfiguration durchgeführt wird und somit Variabilität entfernt wird. Für Aspekte der Bindezeit und deren Auswirkung auf die Codegenerierung sei an dieser Stelle auf [We08] verwiesen.

Nach [JG97] identifiziert ein Variationspunkt „… one or more locations at which variation will occur.“. Diese Stellen werden durch Mechanismen realisiert, die Variabilität entfernen. Über die Blockbibliothek stehen in Simulink eine Reihe von Blöcken zur Verfügung, die zur Realisierung des Variabilitätsmechanismus herangezogen werden können [KB05, TL06]. Beispiele hierfür sind

- bedingt ausführbare Subsysteme (*Enabled*-Subsystem, *Function Call*-Subsystem),
- Signal Routing Blöcke (*Switch*-Block, *Multiport Switch*-Block),
- logische Gatter (*AND*-Block, *OR*-Block),
- konfigurierbare Subsysteme (*Configurable Subsystem*),
- etc.

Jeder dieser Blöcke hat hierbei seinen eigenen Mechanismus zur Steuerung variabler Funktionalität. So wird die Funktionalität, die mittels eines *Enabled*-Subsystems gekapselt ist, je nach Wert des *Enabled*-Signals aktiviert (enabled) oder deaktiviert (disabled). Das *Enabled*-Subsystem eignet sich auf diese Weise besonders zur Modellierung optionaler Funktionalität. Auf ähnliche Weise kann über die logische Verknüpfung mittels *AND*- oder *OR*-Block optionale Funktionalität aktiviert oder deaktiviert werden. Beim Switch-Block wird über das *Control*-Signal eine Variante selektiert – vergleichbar der *‚if-else’*-Kontrollstruktur in C –, wodurch sich der *Switch*-Block insbesondere zur Modellierung alternativer Funktionalität eignet. Im Kontext von Variabilität bezeichnen wir einen derartigen Block als *Resolutionblock*.

Die Mehrzahl dieser Blöcke erfordert ein Eingangssignal, welches die Ausführung des Blocks steuert[2]. Zur Auswahl einer Variante bietet sich hier die Verwendung eines sog. *Kontrollblocks* an. Dieser kann beispielsweise als parametrierter *Constant*-Block oder als *Data Store Read*-Block realisiert sein. Je nach Wert des Kontrollblocks wird die zugehörige Variante ausgeführt.

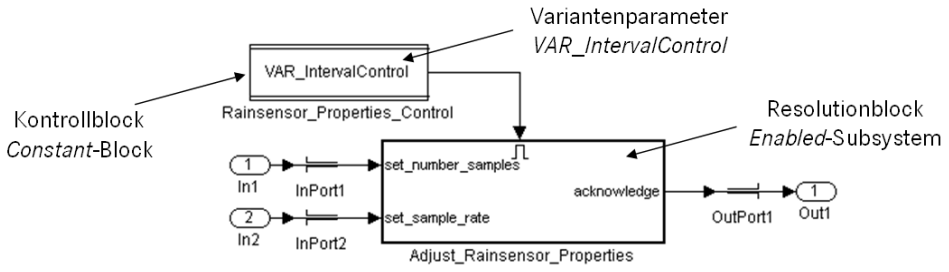


Abbildung 3: Beispiel eines Variabilitätsmechanismus

Abbildung 3) zeigt ein *Enabled*-Subsystem, dessen Ausführung über einen *Constant*-Block gesteuert wird. Der *Constant*-Block kann die selektierte Variante als Wert beinhalten. In diesem Fall wäre der Variantenparameter der Blockparameter *Value* des

[2] Eine Ausnahme bildet hier beispielsweise das *Configurable Subsystem*, dessen Variante über den Wert des Blockparameters BlockChoice selektiert wird.

Constant-Blocks. Falls ein Variationspunkt mehrere Stellen im Modell identifiziert, an denen Variabilität auftritt, empfiehlt sich jedoch ein zentraler Variantenparameter im Model- oder Baseworkspace, der vom Kontrollblock referenziert wird (s. Abbildung 3 der Workspaceparameter *VAR_IntervalControl*).

Um etwaige Kontroll- und Resolutionblöcke im Simulink-Modell variantenspezifisch zu identifizieren und mit zusätzlichen variantenspezifischen Eigenschaften versehen zu können, wird diesen Blöcken ein eindeutiger Maskenparameter, z.B. `var_info`, hinzugefügt[3]. Erst dieser Parameter macht aus einem regulären Simulink-Block einen variantenspezifischen Block. Der Wert des Maskenparameters referenziert den zugehörigen Variationspunkt.

Bedurften die potentiellen Simulink-Blöcke, die als Variabilitätsmechanismus verwendet werden können, jeweils ihre eigene Handhabung, so ist diese nun für alle Variationspunkte gleich. Die Variationspunkte abstrahieren von den verschiedenen Variabilitätsmechanismen in Simulink. Als Datenobjekte gleichen Typs werden sie in einem zentralen Repository gespeichert. Der Zugriff auf die Datenobjekte wurde über eine API, der sog. *Variantenschnittstelle*, realisiert, die auf diese Weise einen uniformen und transparenten Zugriff auf die Variabilitätsinformationen im Simulink-Modell erlaubt[4] (s. Abbildung 4). Diese generische Schnittstelle ermöglicht einen Zugriff beispielsweise über ein entwickeltes Variantenblockset, über die MATLAB-Kommandozeile, über eine Variabilitätssicht, sowie über ein Werkzeug zur Konfiguration von Variabilität im Simulink-Modell.

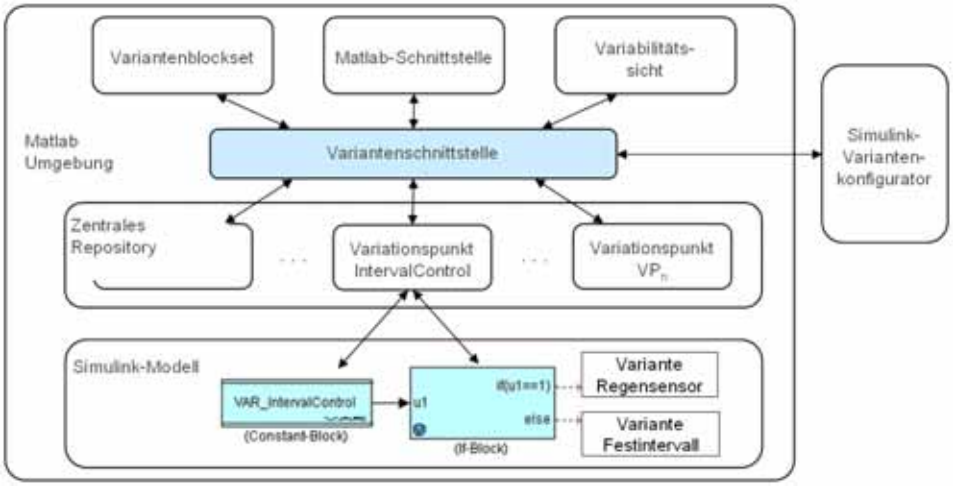


Abbildung 4: Variantenschnittstelle für einen transparenten Zugriff auf Variabilitätsinformation

---

[3] Maskenparameter besitzen im Vergleich zu Blockparametern, wie z.B. *Tag* oder *Description*, den Vorteil, dass diesen ein für die Variabilitätsinformation eindeutiger Bezeichner zugeordnet werden kann.

[4] In unserer Implementierung sind die Datenobjekte im TargetLink Data Dictionary gespeichert. Alternativ können diese auch als Matlab Struktur, Simulink Datenklasse oder Java Objekt gekapselt und als Parameter im Base- oder Modelworkspace instanziiert werden.

3.3 Mapping von Merkmalmodell und Variabilität im Simulink-Modell

Das sog. *Assoziationsmodell* beschreibt die Verknüpfung von Merkmalmodell und Simulink-Modell [KW06]. Sein Konzept und seine Anwendung sind in Abbildung 5 schematisch dargestellt. Das Assoziationsmodell stellt als formale Repräsentation die Abhängigkeiten zwischen den Elementen des Merkmalmodells und den Elementen des Simulink-Modells explizit heraus, und ermöglicht auf diese Weise die Verfolgbarkeit von Variabilität über die Grenzen des Merkmal- und Simulink-Modells hinweg. Dies ist die entscheidende Voraussetzung für eine automatisierte Auswertung der Beziehungen zwischen Merkmalmodell und variantenreichem Simulink-Modell.

Im Wesentlichen beschreibt das Assoziationsmodell eindeutig bei welcher Kombination von Merkmalen welchen Variantenparametern welche Werte aus der jeweiligen Variantenkollektion zugewiesen werden.

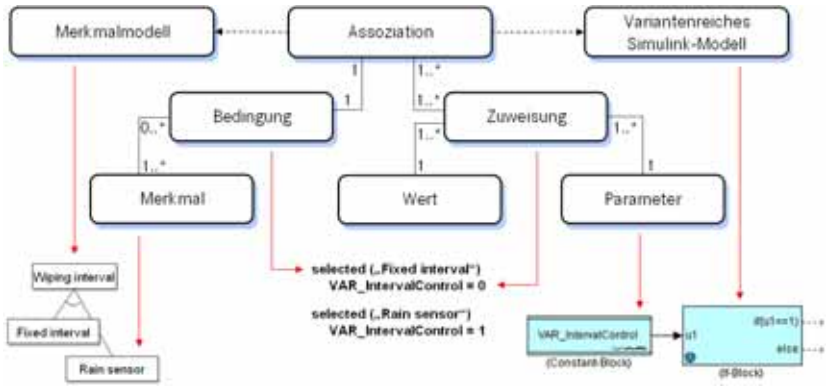


Abbildung 5: Schematische Darstellung des Assoziationsmodells

4 Anwendung der Variantenmodellierung und -konfiguration

Die oben beschriebenen Konzepte erlauben eine systematische Vorgehensweise zur Modellierung, dem Management und der Konfiguration von Variabilität in variantenreichen Simulink-Modellen. Stellvertretend werden nachfolgend die Modellierung variantenreicher Simulink-Modelle im Rahmen des Domain Engineerings und das automatisierte Erzeugen von Modellvarianten aus einer Simulink-basierten Systemfamilie im Rahmen des Application Engineerings näher betrachtet.

Zum Entwurf variantenreicher Simulink-Modelle werden auf Basis des Variantenblocksets Kontroll- und Resolutiotionblöcke in das Simulink-Modell eingefügt (s. Abbildung 6 links unten). Über einen variantenspezifischen Dialog (s. Abbildung 6 rechts unten) werden diese Blöcke anschließend konfiguriert. Der Dialog wird hierbei über eine Open-Callback-Funktion aufgerufen. Im Dialog werden bereits existierende Variationspunkte

zur Auswahl angeboten. Bei einem neu anzulegenden Variationspunkt werden die Informationen des Variantenparameters erfragt:

- Name und Pfad des Variantenparameters,
- seine Wertemenge (die Variantenkollektion) und
- die selektierte Variante.

Zur Darstellung der Variabilitätsinformation existiert eine eigene Sicht auf das Simulink-Modell in Form einer Baumstruktur (s. Abbildung 6 rechts oben). Diese kann nun auf Basis der verfügbaren Information automatisch befüllt werden. Somit wird eine explizite Darstellung der Variabilität im Simulink-Modell ermöglicht. Durch Auswahl eines variantenspezifischen Blocks in der Sicht gelangt man direkt zum entsprechenden Block im variantenreichen Simulink-Modell.

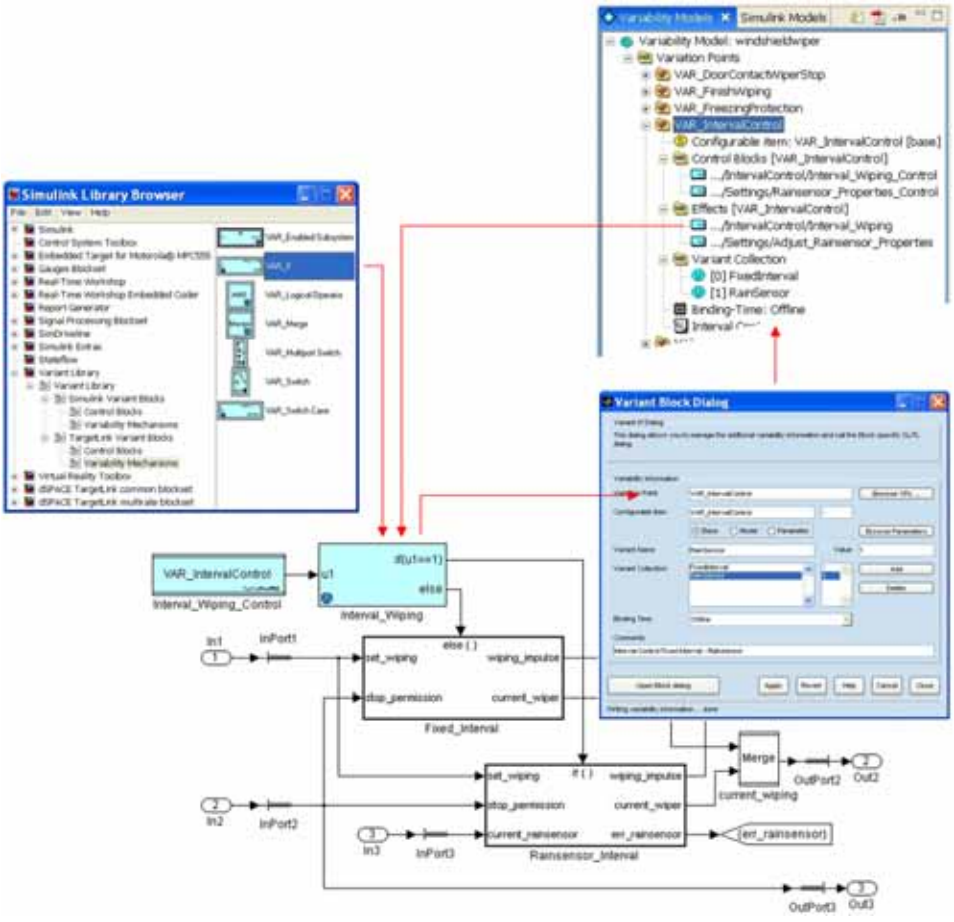


Abbildung 6: Entwurf variabler Funktionen mit Hilfe des Variantenblocksets

Über eine gültige Auswahl von Merkmalen aus dem Merkmalmodell (s. Abbildung 7 links unten) erhält man im Rahmen des Application Engineerings eine Merkmalspezifikation. Mittels dieser kann nun über das Assoziationsmodell (s. Abbildung 7 mitte unten) automatisiert eine konkrete Modellvariante aus dem variantenreichen Simulink-Modell konfiguriert werden. Wesentlich ist hierbei die Vollständigkeit und Widerspruchsfreiheit, d.h. jedem Variantenparameter wird im Zuge der Konfiguration genau einmal ein gültiger Wert aus seiner Variantenkollektion zugewiesen.

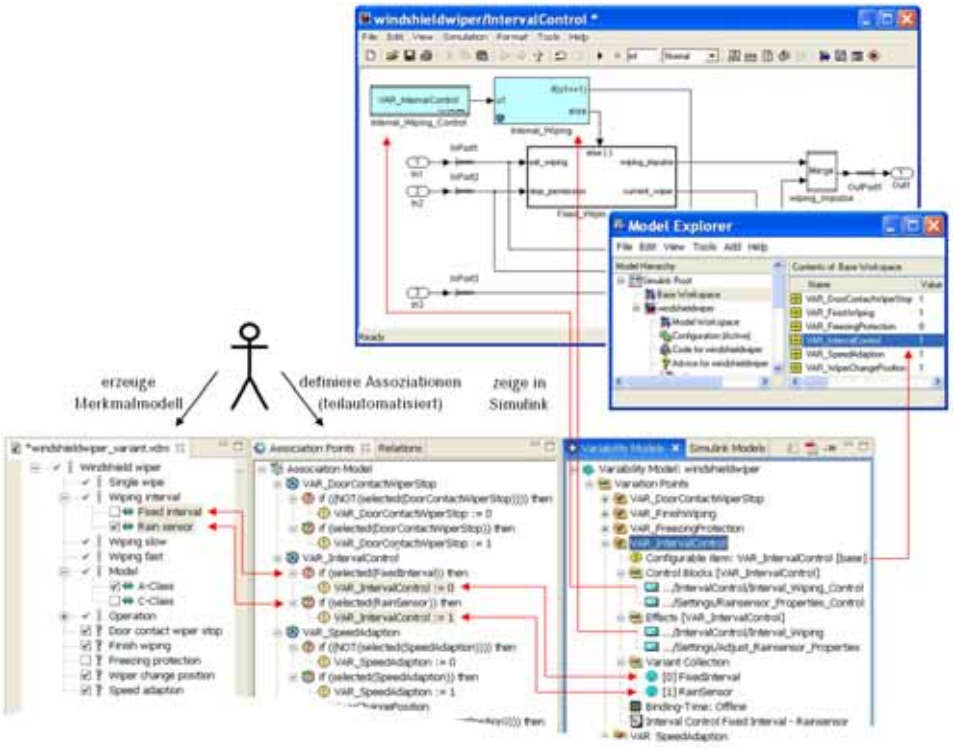


Abbildung 7: Selektives Browsen und Konfiguration von Variabilität in Simulink-Modellen

Für die Implementierung des Simulink-Variantenkonfigurationswerkzeugs wurde hierfür das Tool pure-variants von pure::systems [PS07] integriert.

5 Zusammenfassung

Mit den vorgestellten Konzepten werden bisherige Defizite im Rahmen der Handhabung variabler Funktionalität in Simulink durch eine formale Beschreibung und systematische Modellierung und Konfiguration von Variabilität behoben und somit die Qualität der modellbasierten Software weiter verbessert:

- Die Definition von Abhängigkeiten über das Assoziationsmodell ermöglicht ein selektives Browsen über die verteilte Variabilität im Simulink-Modell; d.h. welche Variationspunkte im Simulink-Modell werden durch welche Merkmale beeinflusst und von welchem Merkmal hängt ein Variationspunkt im Simulink-Modell ab. Auf diese Weise können über das Merkmalmodell valide Spezifikationen erstellt und Konfigurationen, z.B. als Parametersatz oder Konfigurationsanweisungen, automatisiert erzeugt werden, um das Simulink-Modell zu konfigurieren.
- Das Auffinden von Widersprüchen in der Konfiguration von Simulink-Modellen mit komplexen Variabilitäten wird durch die Kopplung der Informationen des Simulink-Modells mit denen des Merkmalmodells erleichtert.
- Variabilität im Simulink-Modell wird explizit sichtbar, einerseits durch die Darstellung von variantenspezifischen Blöcken im Simulink-Modell und andererseits über die Variationspunkte, welche zentral im Simulink-Modell verwaltet und über die Variantenschnittstelle evaluiert werden können.

Die vorgestellten Konzepte wurden im Rahmen eines Forschungsprojektes entwickelt und werden aktuell an einem Serienprojekt der Mercedes PKW-Entwicklung angewendet.

6 Literaturverzeichnis

[Be03] Beuche, D.: *Composition and Construction of Embedded Software Families*. Dissertation, Universität Magdeburg, 2003.

[CE00] Czarnecki, K.; Eisenecker, U.W.: *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CN01] Clements, P.C.; Northrop, L.: *Software Product Lines – Practices and Patterns*. SEI Series in Software Engineering, Addison-Wesley, 2001.

[GS04] Greenfield, J.; Short, K.: *Software Factories – Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[JG97] Jacobson, I.; Griss, M.; Jonsson, P.: *Software Reuse – Architecture, Process, and Organization for Business Success*. Addison-Wesley, 1997.

[KB05] Kalix, E.; Bunzel, S.; Judaschke, U.: *Variant Coding in Model-Based Design*. 9th World Multiconference on Systemics, Cybernetics, and Informatics (WMSCI), Orlando, USA, 2005.

[KW06] Klengel, K., Weiland, J.: *Merkmalbasierte Konfiguration variantenreicher Simulink-Modelle*. In: H. Dörr, T. Klein: Proceedings des Workshops „Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen“, Modellierung 2006, 22.-24. März 2006, Innsbruck, Österreich, 2006.

[LK02] Lee, K.; Kang, K.C.; Lee, J.: *Concepts and Guidelines of Feature Modeling for Product Line Software Engineering*. Software Reuse: Methods, Techniques, and Tools, Vol. LNCS 2319/2002, Springer, 2002.

[PS07] pure-systems, pure::variants Eclipse Plugin User Guide, 2007

[SZ03] Schäuffele, J.; Zurawka, T.: *Automotive Software Engineering – Grundlagen, Prozesse, Methoden und Werkzeuge*. Vieweg, Wiesbaden, Juli 2003

[TL06] dSpace GmbH: *TargetLink Advanced Practices Guide – for TargetLink 2.2*. dSpace, Paderborn, 2006.

[We08] Weiland, J.: *Variantenkonfiguration eingebetteter automotive Software mit Simulink*. Dissertation, Universität Leipzig, 2008.

Using Models for Dynamic System Diagnosis
A Case Study in Automotive Engineering

Oliver Niggemann, Institut Industrial IT (inIT), Hochschule Ostwestfalen-Lippe, Lemgo, Germany,
Benno Stein, Bauhaus University, Weimar, Germany,
Thomas Spanuth, Heinrich Balzer, University of Paderborn, Paderborn, Germany

Abstract: Though various sophisticated concepts for the diagnosis of technical systems have been developed, diagnosis technology in practical applications often boils down to the use of simple heuristics, associative case memories, or manually designed decision trees. These approaches are robust, but restricted with respect to the complexity of the diagnosed system and the faults to be detected.

An inviting direction, which is desired by practitioners and investigated by researchers, aims at the reuse of those models for diagnosis purposes, which were originally designed for system construction and simulation. A promising principle in this connection is model compilation: the model of the interesting system is simulated in various fault modes, and the resulting (huge) set of simulation data is analyzed with machine learning methods, yielding tailored diagnosis rules [Ste03]. To verify this approach, we present a case study from the field of automotive software development. The case study highlights strengths and weaknesses of model compilation. One key challenge is addressed in this paper: the identification of suited symptoms in the data.

Keywords automotive diagnosis, model compilation, simulation, machine learning

1 DIAGNOSIS AND MODEL COMPILATION

1.1 Learning of Diagnosis Algorithms

Diagnosing technical dynamic systems is part of an engineer's core expertise—a task for which engineers rely on their deep knowledge about the application field and about effects of faults in systems. But since systems become more complex it is exceedingly difficult to understand fault impacts and therefore to implement the diagnosis functionalities. This holds true especially for electronic systems in the automotive industry. So from a computer science perspective, a main questions is whether methods from the field of machine learning and knowledge-based systems can be used to support an engineer's work. The goal is not the replacement of human expertise but its formalization, to make it usable by computer algorithms.

Generally speaking, a to-be-diagnosed system can be abstracted as shown in Figure 1.

1. The system itself. The system can either be the real system – e.g. a vehicle – or a simulation of the system. In the later case, which applies to this paper, a system model must exist.

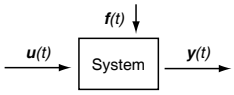


Figure 1: A system stimulated by a vector of input functions $\mathbf{u}(t)$ while failures $\mathbf{f}(t)$ occur. Diagnosis algorithms must resort to the set of measurable signals $\mathbf{y}(t)$.

2. A vector of input functions $\mathbf{u}(t) \in R^k$ defines the driving scenario, such as driving maneuvers and the street situation.

3. During this scenario particular failures (e.g. CAN bit errors), defined by $\mathbf{f}(t) \in \{0,1\}^l$, may occur. A value of 1 at the ith position in $\mathbf{f}$ indicates that the ith failure occurred.

4. The diagnosis algorithm analyses a set of observable variables $\mathbf{y}(t) \in R^m$ to identify the failure causes $\mathbf{f}$.

On which information does a diagnosis algorithm rely, say, what distinguishes a faulty system from an error free system? Which information is required, to decide whether or not the system is faulty? Obviously a system fault can only be identified by comparing the system's behavior to the correct behavior. Such correct behavior might be given as a (simulatable) formalized computer model. This model is often called golden model.

To identify faulty behavior a simple comparison is not enough: Values might differ without constituting significant or critical faults; especially in complex feedback systems such as vehicles random value fluctuations might occur. So differences between golden model and system must be assessed according to their fault significance and criticality. The output of this assessment is a vector of symptoms.

This consideration leads to a model of a diagnosis algorithm shown in Figure 2. A model of an error free system, the golden model, is used to identify incorrect system behavior: The golden model computes the same measurable variables as the system to be diagnosed.

A symptom vector $\mathbf{s}(t)$ is computed from $\mathbf{y_0}$ and $\mathbf{y}$ in a preprocessing step. The symptom vector might contain elements of $\mathbf{y_0}, \mathbf{y}$, or is computed by complex operations on the output vectors. Generally speaking, symptoms correspond to deviations from the normal behavior, or they describe the context in which these deviations occur. For complex systems, identification of symptoms is a major problem and is treated in Section 2. Based on the computed symptom vector a diagnosis algorithm starts (*i*) to explore if a failure occurred in the system and (*ii*) to classify the failure.

Where do the models – golden model and system model – come from? During the development of automotive control software, control and software engineering models are used for specification purposes. These models fall into two classes: (*i*) models for the electronic control units, the ECUs, (e.g. AUTOSAR [AUT] or Simulink®models) that model the behavior and the structure of the ECUs and their respective software, and (*ii*) environment models (e.g. Modelica or Simulink models) that model the behavior of the vehicle (sensors, actuators, engine, gear) and the environment (driver behavior, street conditions). Models comprising both ECU models and environment models are called closed-loop models.

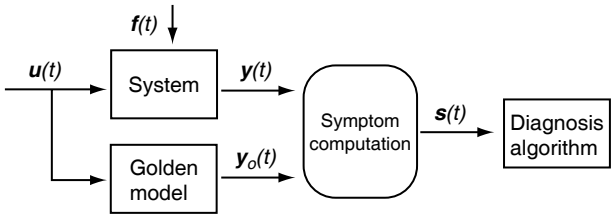


Figure 2: Most diagnosis algorithms rely on knowledge about the system behavior in the non-error case. A golden model is used to compute a set of signals $\mathbf{y_0}(t)$ that are concordant with the measured signals in the no-error case. From a symptom $\mathbf{y}(t)$ and $\mathbf{y_0}(t)$ a vector $\mathbf{s}(t)$ is computed that contains sufficient information to identify the faults.

In this paper those closed-loop models are used as golden models.

The next section will outline the general diagnosis strategy applied in this paper, while Section 2 will describe the key contribution of this paper: A symptom identification approach. This approach is applied in the case study in Section 3. Results are given in section 4.

1.2 The Model-Compilation Approach

In the automotive industry, model-based diagnosis approaches are used quite frequently [FBSI07, SP04, Nyb02, CPD03]. These approaches use symptoms to generate reasonable fault hypotheses, for this the system model is analyzed (see e.g. [Rei87, dKW87]). This generation of fault hypotheses and the ranking of these hypothesis often leads to a non-trivial model analysis problem. Hence, often a specialized *diagnosis model* is constructed by domain experts, where fault deduction is treated in a forward reasoning manner.

The approach applied in this paper is based on a different idea, the so-called model compilation paradigm that was introduced in [Ste01, Hus01]. A salient property of this approach is that neither a too complex analyses of the system model nor a tailored diagnosis model is needed. Model compilation aims at the creation of a fast and precise diagnosis algorithm which is learned as a classification and diagnosis function mapping symptoms onto faults. The learning procedure is based on data recorded during a run of the real system[1] with injected failures and the parallel simulation of the golden model. This empirical data, comprising failures, measurements, and symptoms, is abstracted and generalized in form of the mentioned classification function that *inverts* the causality between failures and symptoms. Such an identification of classification functions is subject of the fields of machine learning and statistics (see [TSK06, Har99] for an overview).

Several model-based diagnosis algorithms use a qualitative model of the system (e.g. in [SS97]) – either generated form the original system model or created manually. Since in this paper model-compilation is used, the system model itself remains unchanged. Instead, the abstraction step is moved to the data mining algorithm. While this approach puts much trust into the machine learning algorithm, it avoids the problem of creating a qualitative

[1] Later on, we will replace recorded data by simulated data.

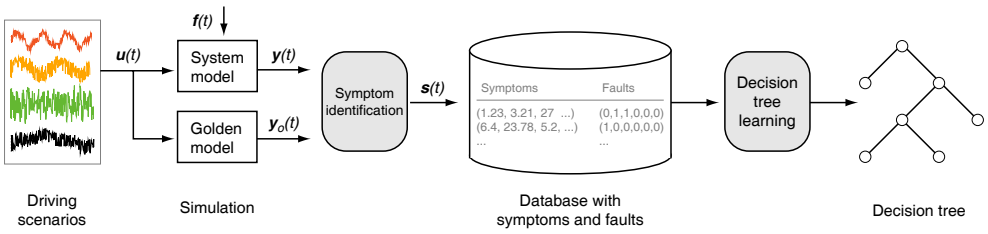


Figure 3: The model compilation process. A large set of driving scenarios are used to simulate the system including faults $\mathbf{f}$ and the golden model. The output vectors of the simulation $\mathbf{y}, \mathbf{y_0}$ are used to generate symptoms $\mathbf{s}$. Domain knowledge is used to generate complex symptoms and form the input for machine learning algorithms to generate a classification tree.

model.

So far, we have not discussed where the data and measurements of the system may come from; especially since for the model-compilation approach a large set of such driving scenarios are necessary. In practice, data about driving scenarios and corresponding faults are hard to obtain; often the faults are unknown or not enough scenarios have been recorded. Beside this it is very expensive to do enough test runs of the real system. A possible way out is the simulation of the real system—in this paper on a PC with an offline simulation. Using this approach, an arbitrary number of scenarios can be simulated and recorded, i.e. huge sets of empirical data can be produced. Based on this empirical data, the classification and diagnosis function is then learned. When such a simulation is used, it is important to verify that the simulation approximates sufficiently the real system.

Since we use simulation to obtain data from the (simulated) "real" system and since in these simulations failures are injected deliberately, the faults $\mathbf{f}(t)$ are known for each point in time. This is a prerequisite for the application of most classification learning algorithms: Since for the learning data the correct classification is known, relations between input data and classification results can be learned. [TSK06] gives an overview of these supervised learning methods. The entire learning process can be seen in Figure 3..

So now besides the golden models for the faultless case, further system models are needed which are able to simulate fault effects, i.e. these additional models are used to simulate the real system. For this, the already mentioned closed-loop models from Section 1.1 can be used also. For automotive diagnosis, the same models are used as golden models for the simulation of the faultless case and as system models for the simulation of the misbehavior. The reader may note that in other domains, golden models and system models might very well be different.

Automotive system models are often able to predict the system behavior in the fault case. For the control part, this follows from the fact that nowadays the software in the ECUs is directly generated from such models, i.e. no significant behavior variations between model and real system are to be expected (see [BOJ04]). For the environment part of the closed-loop model, a general statement is more difficult. But in most cases the models are used during the development to verify diagnosis algorithms, using offline simulation or later in the process using Hardware-in-the-Loop simulation (see [ONS+07]). Note that

these environment models therefore have to cover the prediction of fault effects since they are used for the testing of diagnosis algorithms. Hence, the environment models can be used to simulate fault effects in most cases.

Finally, in this paper decision trees [BFOS84] are used as classification function since they are applied on a frequent basis by engineers for automotive diagnosis. Automotive diagnosis happens at different stages of the vehicle life-cycle: within the vehicle, during the development, and in the garage. For the later case – which is the focus of this paper – decision trees are used in most tool chains, i.e. other approaches could not be integrated into existing working processes.

2 THE PREPROCESSING STEP: GENERATION OF SUITED SYMPTOMS

The model compilation approach from section 1.2 has already been applied successfully to various problems in the past—e.g. in [Ste01, Hus01]. But its key advantage—using machine learning to inverse the *fault→symptom* causality which allows for the usage of existing models—also causes its main weakness: Much trust is put in the power of machine learning algorithms. In this section that weakness is addressed by means of preprocessing steps that compute a set of high-level symptoms which eases the machine learning task.

The reader may ask why such a preprocessing step is necessary. Looking at Figure 3, it becomes clear that theoretically all information needed by the learning algorithms is already part of $\mathbf{y}$ and $\mathbf{y}_0$. For two main reasons a preprocessing step may significantly improve the model compilation approach: (*i*) Engineers often apply domain-depended data preprocessing steps which can not be identified automatically. This domain knowhow should be provided to the machine learning algorithms. (*ii*) Most machine learning algorithms have problems with learning complex, e.g. nonlinear or time-dependent, functions: Either they are only able to handle a limited set of rather simple function patterns or they support complex functions but rely for that on non-deterministic and time-consuming optimization methods—details can be found in [TSK06]

In detail, machine learning algorithms face three main problems when applied to such diagnosis tasks:

1. The diagnose function need not make decisions based on discrete values because otherwise only remembered old situations could be treated correctly. Instead the diagnosis function must abstract from discrete values, i.e. it must use value ranges and other complex value features. That way, new so-far unknown situations can be classified correctly.

2. The measured and simulated values are functions over time. These values must be transferred into stationary, i.e. time independent, features.

3. Often value combinations must be used to classify faults successfully. Such value combinations can be identified in a preprocessing step.

This section will therefore first present some symptom abstraction methods which either

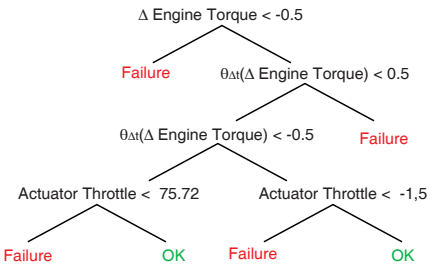


Figure 4: Decision tree for signal interruption of the speed sensor.

are computed automatically or are computed manually by engineers—this takes care of point 1 from above. Next, algorithms for generating stationary symptoms are presented.

2.1 Value Abstraction

The most important value abstraction step in this case study is done automatically by the decision tree learning algorithm. The algorithm used (REPTree and J48 from the Weka-Library, [WF05])) splits values into value ranges. Figure 4 shows an example: Each decision splits the value range of one variable, i.e. the decision is based on value intervals instead of discrete values. Because this step is done inherently by the machine learning algorithm, value ranges need not be computed explicitly in a preprocessing step.

We found that in practice the following preprocessing operations are often applied by engineers to diagnose failures in automotive systems; each of these steps computes abstracted value features:

Generation of Deviations from the Golden Model. The simplest operation in this category is the computation of the difference between the output of the (simulated) real system and the golden model: $\mathbf{y} - \mathbf{y}_0$. This makes the diagnosis function independent of the absolute values.

Generation of Statistic Characteristics. Important symptoms can be based on statistical properties of $\mathbf{y}(t)$:

- The standard deviation of $\mathbf{y}(t)$ carries valuable information about the amount of noise in the data. The standard variation σ is normally computed for the last Δt time steps: $\sigma_{\Delta t}(\mathbf{y}(t)) = \sqrt{E(X^2) - (E(X))^2}$ with $X = \mathbf{y}(t-\Delta t), \ldots, \mathbf{y}(t-1), \mathbf{y}(t)$. The same operations can of course be applied to $\mathbf{y}_0$.

- A very helpful symptom is the information whether a value $\mathbf{y}(t)$ is still "normal". The term "normal" is defined by the probability that the measured value $\mathbf{y}(t)$ is still within the normal variance of $\mathbf{y_0}(t)$: $P_{\Delta t}(\mathbf{y}(t)) = \int_{\mathbf{y}(t)}^{\inf} n(\mathbf{e}_{\Delta t}(\mathbf{y_0}), \sigma_{\Delta t}(\mathbf{y_0}))$ where $\mathbf{e}_{\Delta t}$ denotes the expected value and $\sigma_{\Delta t}$ the standard variance for the data vector during the last Δt time steps. n is the Gaussian probability distribution.

  This can also be seen in Figure 5: The probability that a measurement y is still "normal" is defined by the integral under the Gaussian distribution of the golden

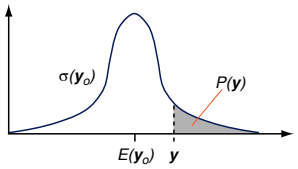


Figure 5: The probability of a measurement from the real system y to be "not normal" depends on the distribution of that measurement in the golden model, i.e. of y_0.

model y_0. $e(y_0)$ denotes the expected value of y_0 while $\sigma(y_0)$ denotes the standard variance.

- The quotient $\frac{\sigma_{\Delta t}(\mathbf{y}(t))}{\sigma_{\Delta t}(\mathbf{y_0}(t))}$ denotes the increase of noise in the data and is therefore a valuable symptom for noise related faults.

2.2 Stationary Symptoms

Only few machine learning algorithm can directly use temporal data (e.g. [CPD03])—none of which have established themselves. The easiest way to handle temporal data is to make the data stationary, i.e. time independent. E.g. the time-dependent function *vehicleVelocity*(t) is replaced for one diagnosis scenario by an approximation v, v_dev (modeling the current speed and the first current derivation). Here, a two step approach is used:

Step I: Mode Identification. Diagnosis functions assume that the causal relation between failure causes and failure effects does not chance between the generation of the diagnosis rules and the usage of those rules during a vehicle's lifetime. These relations remain stable only under specific conditions — e.g. similar driving situations. Such a situation, that can be addressed by one set of diagnosis functions, is called a mode. In this example, modes correspond to one gear of the vehicle, i.e. the mode can be identified in a rather straight forward way.

Algorithms for the automatic identification of modes can be found in [Ste01, Hus01].

Step II: Generation of Temporal Stationary States. Within each mode, stationary features are generated from time dependent data. For this, diagnosis algorithms often use the history of the vectors $\mathbf{y}(t), \mathbf{y}_0(t)$: An increasing position of the gas pedal should result in an increasing vehicle speed. If the vector values are only used at one specific point in time t, such correlations can not be exploited. Several methods exist to capture the history:

- The history $\mathbf{y}(t_1 \dots t_n)$ can be captured by the following function $\theta_{\Delta t}(\mathbf{y}(t)) = \mathbf{y}(t) - \mathbf{y}(t - \Delta t)$. The identification of a suited Δt is a challenge by itself. Here an initial Δt has been guessed based on system properties, during the machine learning process an optimization method has been used to find an optimal value.
- While the previous function is a linear approximation of $\mathbf{y}$'s history during the last Δt time steps, more complex approximations—e.g. using polynomial approximation functions—can be applied.

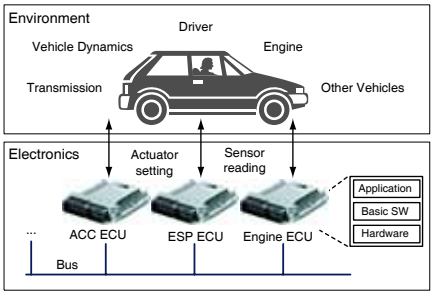


Figure 6: A coarse ACC architecture with the differentiation between ECU models (bottom) and environment models (top).

The same operations are of course applied to $\mathbf{y}_0$.

3 AN EXAMPLE: ADAPTIVE CRUISE CONTROL

3.1 The Adaptive Cruise Control Application

An Adaptive Cruise Control (ACC) is used in modern vehicles to control automatically the distance to cars driving ahead in the same lane. If the car in front of the driver slows down, the ACC will also decrease the vehicle's speed by reducing the throttle setting or even by activating the brakes. If the traffic situation allows for an increase of the vehicle's speed, the ACC will resume again the predefined cruising speed. In order to measure the distance to the up-front traffic, radar, laser, and optical sensors are used.

To implement the ACC functionality, several electronic control units (ECUs) play together: (*i*) Usually the sensor interpretation and sensor fusion happen in a special ACC ECU. (*ii*) Braking activities are often controlled by an Electronic Stability Program (ESP®) ECU. (*iii*) Engine control including throttle position control reside in the engine ECU. (*iv*) User interactions such as turning on the ACC or setting the cruising speed are handled by the central ECU or the display ECU.

All these ECUs are connected via one or several communication buses such as CAN or FlexRay. This architecture is illustrated in Figure 6. Details about the ACC functionality can be found in [Gmb03].

Figure 7 shows the model used in this example: Here Simulink has been used to model both ECU software (left-hand side) and simple models for the engine and ESP ECU (right-hand side). The environment side also comprises models for sensors and actuators.

3.2 ACC-relevant Faults

Figure 7 also pinpoints the main fault sources within ACC systems. The control algorithms in the ECUs (first column in Figure 7) rely on environment information delivered by the sensor layer (rightmost column). Furthermore, the control algorithms use the actuators to control the environment. Hence, four main fault locations can be identified:

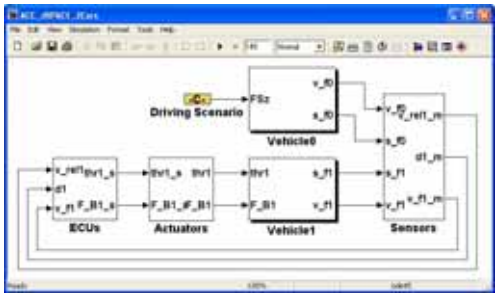


Figure 7: A Simulink Model of an ACC. The ECU models are bottom left. The output of these models forms the input for the actuator models (2nd column) which are directly connected to the environment model of the vehicle ("Vehicle1" in the 3rd column). The subsystem "Vehicle0" models the behavior of the car ahead. On the right, the sensor models can be found that measure data in the vehicle and transport them to the ECU models.

1. Signals coming into the sensor layer could be faulty. This may correspond to faults in the vehicle's wiring.
2. Signals leaving the sensor layer could be faulty. This could either model faulty sensors or a faulty interpretation and preprocessing of the sensor data by the ECU software.
3. Faulty signals coming into the actuator layer correspond to faulty control algorithms or to faults in basic software modules such as I/O drivers.
4. Faulty actuators (including the corresponding wiring) correspond to faulty signals leaving the actuator layer.

In our case study we concentrate on signal faults at these four locations. Faults can be diagnosed at different point in times: While the car is driven, in the garage, or during the vehicle's development. Here we mainly target the second scenario.

4 DIAGNOSIS RESULTS

In cooperation with a leading automotive tool provider and modeling company, the diagnosis method presented in this paper has been used to diagnose the following faults in the ACC system from section 3: (*i*) A faulty throttle actuator, (*ii*) a faulty brake actuator, (*iii*) a faulty computation of the velocity of the vehicle ahead, and (*iv*) a faulty engine crankshaft speed sensor.

For each faulty component four fault subtypes have been addressed: $\mathbf{1}$. Additional noise on the signal, $\mathbf{2}$. 10% positive offset onto the signal, $\mathbf{3}$. 10% negative offset onto the signal and $\mathbf{4}$. total loss or interruption of the signal.

Fault i with fault subtypes j is in this paper denoted as $f_{i,j}$. The term f_i denotes the fault i without further differentiation between fault subtypes. All error rates are measured on test sets that have not been used to learn the decision tree. In this paper a separate classification

Fault	f_3	f_4	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$
Error Rate	14%	12%	16%	13%	9%	1%

Table 1: Typical Error Rates

tree t_i is learned for each f_i, i.e. $t_i : \mathbf{s} \rightarrow \{0, f_{i,1}, f_{i,2}, f_{i,3}, f_{i,4}\}$, where 0 denotes the diagnosis "no fault". Classification functions are learned separately for each gear setting since gear settings form different modes of the underlying mathematical models. So in a later diagnosis setting, the overall diagnosis function will choose the correct classification tree depending on the actual gear.

So all error rates given in this paper denote the percentage of incorrectly diagnosed situations whereat the diagnosis rules have been learned using scenarios different from those later used to measure the error rate. The driving scenarios include different acceleration and deceleration situations; in these situations all gear settings are used.

For the learning itself the REPTREE and the J48 algorithms from the WEKA library (see [WF05]) have been used. The error rates given for such a decision tree always take into consideration that faults $f_{i,j}$ have to be distinguished from other faults $f_{k,l}$ ($k \neq i \vee l \neq j$)—i.e. the learned classification functions have to separate between the different faults.

Typical results can be seen in Table 1.

The learned decision trees are also meaningful from an engineer's perspective: Figure 4 shows an example of a learned decision tree for a signal interruption at the engine speed sensor. Taking knowhow from the ACC domain into consideration, this tree makes sense because the actuator for the throttle position receives its values against the engine speed. Moreover there is a direct relation between engine torque and engine speed. So a faulty engine speed sensor would be noticeable in the relation between engine torque and engine speed which is exactly what has been exploited by the learned decision tree.

5 SUMMARY

In this paper a case study for the diagnosis approach of model compilation is given. The case study is used to outline open challenges and to give first solutions. One focus of this paper is the challenge of symptom identification. Methods for symptom identification have been tested using the diagnosis of an Adaptive Cruise Control (ACC) as an example. It has been shown that sound classification trees can be learned using this approach. The application of model compilation would enable engineers to leverage on their existing system models to implement diagnosis functions for increasingly complex systems and might render specific diagnosis models unnecessary.

References

[AUT] AUTOSAR. Internet Homepage www.autosar.org.

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen und C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[BOJ04] M. Beine, R. Otterbach und M. Jungmann. Development of Safety-Critical Software Using Automatic Code Generation. *Society of Automotive Engineers (SAE)*, 2004.

[CPD03] L. Console, C. Picardi und D.T. Dupre. Temporal Decision Trees: Model-based Diagnosis of Dynamic Systems On-Board. *Journal of Artificial Intelligence Research*, 19:469–512, 2003.

[dKW87] Johan de Kleer und Brian C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[FBSI07] D. Fischer, M. BÃűrner, J. Schmitt und R. Isermann. Fault detection for lateral and vertical vehicle dynamics. *Control Engineering Practice*, 15(3):315–324, 2007.

[Gmb03] Robert Bosch GmbH. ACC Adaptive Cruise Control. The Bosch Yellow Jackets, 2003.

[Har99] Jens Hartung. *Statistik*. Oldenbourg, 1999.

[Hus01] Uwe Husemeyer. *Heuristische Diagnose mit Assoziationsregeln*. Dissertation, University of Paderborn, Department of Mathematics and Computer Science, 2001.

[Nyb02] Mattias Nyberg. Model-based diagnosis of an automotive engine using several types of fault models. *IEEE Transaction on Control Systems Technology*, 10(5):679–689, September 2002.

[ONS+07] Rainer Otterbach, Oliver Niggemann, Joachim Stroop, Axel ThÃijmmler und Ulrich Kiffmeier. System Verification throughout the Development Cycle. *ATZ Automobiltechnische Zeitschrift*, 2007.

[Rei87] Raymond Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[SP04] Peter Struss und Chris Price. Model-based systems in the automotive industry. *AI Magazine*, 24(4):17–34, 2004.

[SS97] Martin Sachenbacher und Peter Struss. Fault Isolation in the Hydraulic Circuit of an ABS: A Real-World Reference Problem for Diagnosis. In *Working Papers of the Eighth International Workshop on Principles of Diagnosis (DX-97), Mont Saint Michel, France*, 1997.

[Ste01] Benno Stein. *Model Construction in Analysis and Synthesis Tasks*. Habilitation, Department of Computer Science, University of Paderborn, Germany, Juni 2001.

[Ste03] Benno Stein. Model Compilation and Diagnosability of Technical Systems. In M. H. Hanza, Hrsg., *Proceedings of the 3rd IASTED International Conference on Artificial Intelligence and Applications (AIA 03), BenalmÃądena, Spain*, Seiten 191–197. ACTA Press, September 2003.

[TSK06] Pang-Ning Tan, Michael Steinbach und Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2006.

[WF05] I. H. Witten und E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

Domain-specific Modeling, Validation, and Verification of Railway Control Systems

Kirsten Mewes

Verified Systems International GmbH
Parkstrasse 123
28209 Bremen
kirsten@verified.de

Abstract: Railway control systems are usually large systems that are tailored to specific implementations by configurations. The large number of possible configuration data leads to difficulties in verification and validation as the combinations of different configured values and their impact on the system behavior are hard to assess. The degree of automation is low, especially for validation. In this paper, we show a new approach where small controllers designed for a specific application are developed by means of a domain-specific modeling language. This language incorporates domain-specific knowledge which is then used in the validation and verification process. We show how models can be automatically validated, used for automated code generation of simulations, and used for test data generation. The high degree of automation helps reducing development costs while at the same time the foundation on domain-specific knowledge influences the product quality positively.

1 Introduction

The development of railway control systems is a challenging process. One reason for this is that the railway domain meets software engineering, two very different disciplines. These have to co-operate to achieve high quality results. The domain experts have to write down the requirements for the software in a way that they are implementable. At the same time, the software engineer has to understand these domain-specific requirements to implement them correctly. The high need for product quality is beyond dispute as human life may be endangered if a railway controller is malfunctioning. The struggle for high-quality software development methods is of highest importance.

Currently, most controllers are developed as a generic piece of software that is configurable. A concrete system is hence a projection of the generic controller. This kind of development is established but often criticized. Arguments against this proceeding are manifold: unnecessary large software, hard maintenance, and hard to test which results in difficulties in *validation* and *verification*. The alternative is the development of a small system for each projection which is easier to maintain. However, all validation and verification tasks have to be performed for each system while they have to be performed just once for the generic system which is the reason for its establishment [PBD+05].

Validation and verification are tasks that are mandatory for the development of railway systems as regulated by the relevant norm *CENELEC 50128* [CEN]. As they consume time and are hence costly, one goal in software development for railway controllers is minimizing the costs for these tasks while guaranteeing high quality at the same time. A means to achieve this goal is automation by using new techniques in software development. Especially interesting are *domain-specific languages (DSLs)* as these are able to incorporate domain knowledge in a small but expressive language.

In the following sections, we show how a DSL for the railway control systems domain (RCSD) is constructed and how its strengths are used to optimize validation and verification. Each controller is modeled by its domain of control - a track layout - and the used components. The model is validated against domain-specific rules such as the consistency of routes of the track layout. Further tasks rely on this validated model. First, controller code can be generated for simulation purposes. Second, test data is generated that can be used as input to the simulation and also further for system integration tests.

We discuss DSLs shortly in Sec. 2. In Sec. 3, the railway control systems domain is introduced. The specification of the DSL is examined in Sec. 4, while validation and verification are handled in Sec. 5. The achieved results are discussed in Sec. 6.

2 Domain-Specific Languages and their Design

Domain-specific languages are not a completely new idea, they have been around for a while as can be seen e.g. in [Ben86] or [vDK98]. The basic idea is having a simple and small language that is expressive and comfortable to use in a specific domain. Contrary, a general purpose language (GPL) is large and unspecific but applicable in every domain. Examples for programming languages are DOT [Dot] as a domain-specific language for defining graphs and C as a general purpose language. Apparently, graphs can be implemented in C, but using DOT is the more efficient approach.

The advantages of DSLs are obvious: a small language is easy to learn and use, and efficient in its context as the concepts of the domain are directly accessible and domain-specific notation can be used. As a consequence, engineering costs in software development are reduced [vDK98]. Obvious drawbacks are the development and maintenance costs of the DSL: Like every language, a DSL consists of abstract syntax, concrete syntax, and semantics. A domain analysis has to be performed to identify the domain-specific concepts, relationships, and rules. As a result, DSL design is time consuming and costly, hence, DSLs have never been a popular design paradigm in software development.

The interest in modeling languages like UML and visual modeling tools has changed this situation. The UML has been criticized much due to its immense size and complexity. This lead to the development of DSL frameworks such as MetaEdit+ [Met] or GME [GME] that support the design of visual modeling languages and hence facilitate DSL design. The possibility to develop a DSL efficiently lead to an increased usage of the approach in industry [KT08]. In the following, we will examine the impact of an appropriate DSL in the safety-critical railway control systems domain.

3 Railway Control Systems Domain (RCSD)

As the first step in the development of the railway control systems DSL, we have to perform a domain analysis. Relevant concept, relationships, and rules have to be identified. Also, the boundaries of the domain have to be fixed to keep the language as simple and small as possible. One can study the domain by consulting books [Pac02] and domain experts. In our case, the domain is small as we are focusing on the development of controllers. Necessary concepts are the parts of the track such as segments or signals, as well as route definitions that describe the ways a train can take on the track. Mechanical details like different kinds of rolling stock are irrelevant as the controller is not aware of them. Another restriction is that we consider only main tracks as side tracks are usually driven by sight and not controlled automatically.

As a result, we have identified seven basic concepts: sensors that inform the controller about train movements on the track, segments, crossings, and points as track components, signals and automatic train protection systems (ATPs) as additional track equipment, and routes. Points, signals and ATPs are the controlled items, while sensor states and route requests are the basic inputs. Additionally, signal, point, and ATP states are considered as input for detecting malfunctioning. Tracks are composed to track layouts.

Each of these concepts has to be further investigated to examine details and relationships. As an example, detailed information is needed for each sensor: the current state (`LOW` or `HIGH`), the timestamp of the last state change, a counter detecting passing trains, and an identification number for addressing the sensor. Sensors are related to track components where they detect if a train is entering or exiting. They are also related to signals and ATPs as these components are positioned at sensors.

Furthermore, the rules of the domain have to be captured. One simple example is that a train may not pass a signal showing `HALT` which we can detect by comparing sensor state changes to a set of expected sensor state changes at each point in time. The correct handling of this and similar situations is that the controller tries to establish a failsafe state which means setting all signals to `HALT` and activating all ATPs that in turn trigger breaking for all passing trains. More details can be found in [BH06] and [Ber07].

4 RCSD DSL

As already described in Sec. 2, an appropriate means for designing a DSL is using a DSL framework. MetaEdit+ has been chosen from the available tools as it offers a metalanguage for specifying abstract syntax and powerful support for defining concrete syntax. In addition, the generator mechanism provides possibilities for checking static syntax rules and code generation. In the following, some excerpts from the domain-specific RCSD language are explained.

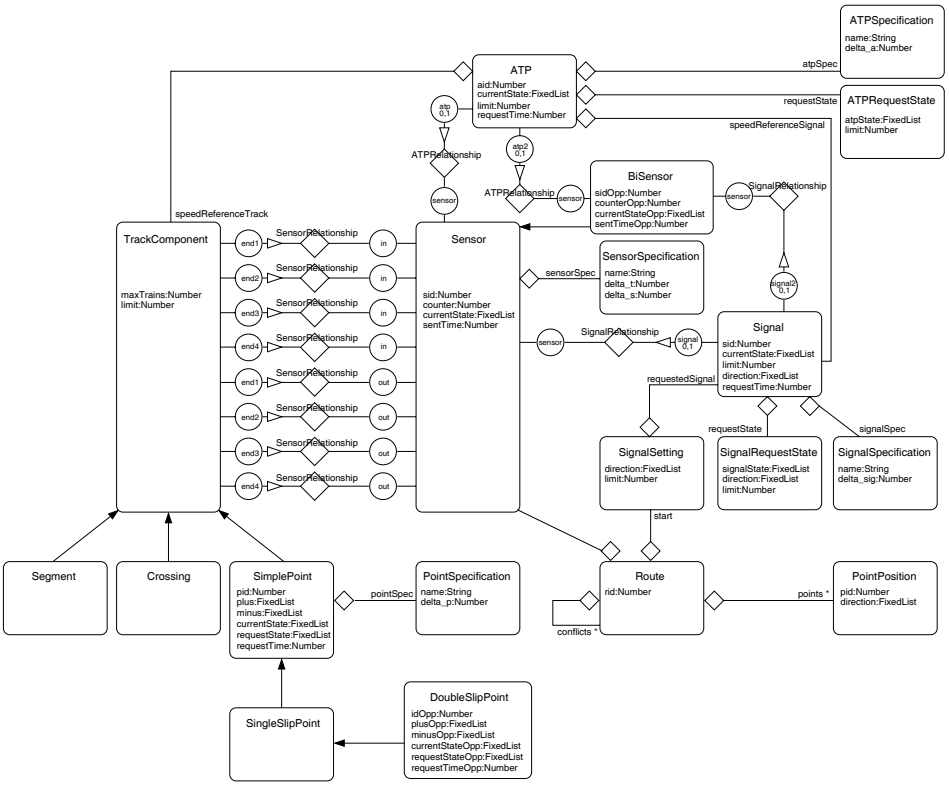


Figure 1: RCSD abstract syntax

Abstract Syntax The abstract syntax of the RCSD language is modeled in the metalanguage of MetaEdit+ which provides concepts for graphs, objects, properties, relationships, and roles in relationships. The language constructs that have been identified in the domain analysis described in Sec. 3, their properties, and relationships are visualized in Fig. 1. Here, we can see e.g. that there are uni-directional sensors modeled as object `Sensor` and derived bi-directional sensors `BiSensor` with the properties and relationships that have been already discussed. We can also see that `SensorSpecification` has been additionally defined to provide information about different kinds of sensors with different characteristics. Most important are the specified latencies `delta_s` and `delta_t`. The first one describes the time a sensor must be in state `HIGH` to detect a train, while the second one configures the time that must pass to detect two subsequent trains independently.

Concrete Syntax The concrete syntax of the RCSD language is visualized in two diagrams. The first of these is a `Specification Diagram` as visualized in Fig. 2. Here, information about the available components is presented, i.e. the different kinds of sensors, signals, points, and ATPs. In this example, two kinds of sensor with different latencies are

present and one kind of point, signal, and ATP each.

The second kind of diagram is the `Track Layout Diagram` that has to be associated to a specification graph as each sensor, signal, point, or ATP is linked to one element defined there. The track layout visualizes the concrete track that has to be managed by a controller. In Fig. 3, an example consisting of several segments, sensors, signals, and points is shown. Track segments are visualized by lines that are connected by sensors, e.g. sensor `S14` at the leftmost end of the upper track. Signals are connected to sensors as we can see for signal `Sig1` that is associated to sensor `S1`. A example for a point is `P4` which is used to unite the upper and lower track in the figure to one track.

Two routes `R1` and `R2` are defined which can be observed beneath the tracks. Each route description consists of a sequence of sensors describing the route, a set of requested point positions so that the route can be driven safely, the start signal, and a set of conflicting routes that may not be released at the same time for safety reasons.

Figure 2: Specification diagram example

Static Semantics There are two ways of specifying static semantics in MetaEdit+. First, there is a set of predefined rules that can be utilized for each kind of graph. Second the generator mechanism can be used by writing a static semantics checker. An example for predefined rules are *uniqueness constraints* where values of properties in a graph are restricted to unique values. A typical example is the sensor whose identification number `sid` has to be unique in each graph.

More elaborated rules have been specified with the generator. The rules are automatically checked on request and violations are reported. The basic principle of the generator is that is loops through the set of instances of each object, e.g. all instances of `Segment`. For each object instance, properties can be examined and relationships can be followed. In this way, the generator crawls through a model. A simple example is shown in the following. For each `Segment`, it has to be ensured that two sensors are connected. The reason for this is that we have to detect the positions of trains on the track.

This rule looks as follows:

```
$count = '0'
do >SensorRelationship
{
   $count++%null
}
if ($count <> '2' num) then
  'Problem with segment: ' id newline
  'Each segment must be connected to two sensors.' newline
  $++errors%null
endif
```

This piece of code is processed for each `Segment`. A counter is incremented for each attached `SensorRelationship`. We can assess the number of connected sensors as there is one at the other end of each `SensorRelationship`. Note that `%null` does not refer to a modulo operation, but suppresses the output of the variable value. An output is generated in case of an error. The `id` is a property of `Sensor` that has been marked as internal identification number. In this case, it is identical with the sensor identification `sid`. The id value is automatically printed out as accessible link that leads to the erroneous element in the model. Hence, problems can be easily backtracked. A more interesting example is to check if the set of conflicting routes includes all necessary routes to ensure safety. This includes the routes that require a different point position for a specific point, the routes that use the same crossing in different directions, and the routes that use some track component in opposite directions which would lead to collisions.

Dynamic Semantics Dynamic semantics have been specified as a timed state transition system (TSTS) [BH06]. The set of elements of our domain E is given by the elements in a RCSD track layout, i.e. the sensors, points, signals, ATPs, and routes: $E = Sens \times Pts \times Sigs \times Atps \times Rts$. Each element has a variable and a constant part, e.g. the current state `currentState` of a sensor is a variable and the associated latency `delta_s` is a constant. The variables of an element are denoted as $var(e)$, hence the variables of all elements are $\bigcup_e var(e)$. In addition, we have the time t and the indicator $FAIL$ which is used to signal the failsafe state. Consequently, our system consists of the variables $VAR = \bigcup_e var(e) \cup FAIL \cup \{t\}$

A state σ in our TSTS is defined as a type-consistent mapping from variables to values. The set of all states is called Σ. $\sigma_0 \in \Sigma$ is the initial state that assigns a value to each variable. Transitions tr lead from a state σ to another state σ'. The set of all Transitions is TR. Note that transition tr can also be expressed as $\sigma \rightarrow \sigma'$. As a result, our TSTS SYS is defined as $SYS = (\Sigma, \sigma_0, TR)$.

As an example transition, we will have a look at sensors again. Each sensor has a current state that can be `HIGH`, `LOW`, or `FAILURE`. A state change from `HIGH` to `LOW` is valid if the sensor was at least `delta_s` units in state `HIGH` and the `FAIL` variable does not indicate the failsafe state. The reason for this rule is that we can expect a minimal indication time for each train on the sensor. It this is not exceeded, the sensor is probably not

working correctly. We assume that glitches are suppressed by the sensor interface, hence, they are not considered here.

The corresponding transition looks as follows:

$$\begin{array}{ll} \sigma' & = (\sigma : sens.currentState \mapsto LOW) \\ pre & \sigma(sens.currentState) = HIGH \wedge \sigma(t) > \sigma(sens.sentTime) + \\ & sens.delta\_s \wedge \neg FAIL \end{array}$$

5 Validation and Verification

Verification and validation are tasks that are mandatory according to the *CENELEC* standard [CEN]. *Verification* is performed at each step of the software development process. The goal is to verify that the requirements that have been defined in the development step before are fulfilled in the current step. Examples are the verification of code which includes the correct application of coding standards, the verification of modules which includes module tests against the module specification, or software and hardware/software integration tests that check that the specified requirements are met. Each step is separately documented. *Validation* is the overall task that checks that the integrated system fulfills its requirements. The main focus is on safety-critical aspects and the functionality of the software. Typical means for validation are analysis and testing. Simulations and models may be used as complementary methods. Tests performed during verification may be referenced and reused for validation.

With respect to verification, we will examine testing and here, especially automated test case generation. Note that formal verification techniques as e.g. model checking can also be used. However, this is only a recommended technique in the standard while testing is mandatory. The generated test suite focuses on the normal behavior of the system as well as the correct handling of safety-critical situations. As a result, the test suite is applicable for software and hardware/software integration testing in the verification process as well as for validation purposes. In addition, the model that serves as basis for the controller is automatically validated against the domain-specific rules. Validation is further supported by the possibility to simulate the control system. Moreover, the applicability of the generated test suite can be demonstrated by mutant testing. A simulation environment supports testing on model level and mutant testing.

Model Validation For models designed with the RCSD language, we have two kinds of validation. First, the inherent validation that takes part due to the language definition in MetaEdit+. The predefined rules that we have utilized as described in Sec. 4 are automatically supervised. It is not possible to create e.g. two sensor instances with the same identification number. The explicit validation is based on the static semantics checker defined with the generator mechanism. In Sec. 4, we have already discussed that each route has an associated set of conflicting routes that has to be checked as collisions have to be avoided. In Fig. 3, we can see an example where this rules fails. The reason is that `Route 2` does not include `Route 1` in its set of conflicting routes. This is compulsory as `Route 2`

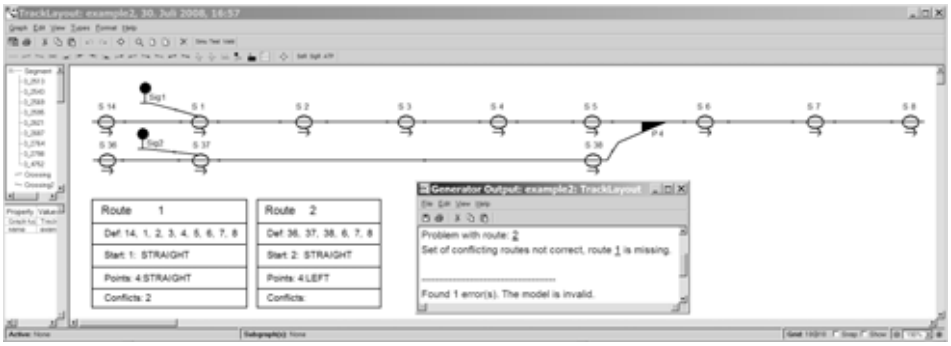


Figure 3: Track layout example with validation output

requires `Point 4` in position `LEFT`, while `Route 1` expects position `STRAIGHT` from the same point. Hence, it is impossible to schedule these two routes at the same time.

Testing A high degree of automation is required to reduce costs for testing. Hence, we have to develop a strategy for test automation that is compliant to the CENELEC standard. Algorithms for model-based generation of test data for timed systems exist (see e.g. [CO02]), but they are not easily applicable to non-deterministic systems like RCSD as they are based on finding paths in the transition system describing the dynamic semantics. In our case, most paths will lead to the failsafe state as normal behavior relies heavily on route definitions and the knowledge about conflicting and non-conflicting routes. This information is not directly accessible in the TSTS. However, CENELEC demands that the applicability of the test suite is motivated in the validation process which includes testing normal and exceptional behavior, the latter in particularly for safety-critical situations.

Hence, we propose to use the domain knowledge for the selection of test data. Both the normal behavior and the safety-critical situations that have been defined in the domain analysis are considered. In the first case, examples are test cases that cover that a route can be traveled, that a route can be traveled at the same time as routes that are not in conflict, that routes that are in conflict can only be traveled subsequently, and that the maximal number of trains allowed on a route can be dispatched (and not more). Safety-critical situations form the second block of generated tests data: Here we have to check that each problematic situation is detected and the failsafe state is reached. This includes e.g. failing elements, disobeyed signals, and unexpected sensor state changes.

All test cases can be generated as the necessary information like route definitions or latencies - and hence valid and invalid time intervals for state changes - can be extracted from the model. As an example, 72 test cases are generated for the model shown in Fig. 3. A small example is given below: Here, the first route is requested at timestamp `0` which requires `Point 4` in the correct position `STRAIGHT`. This request to the point is ensured at timestamp `1`. The signal is set in time at timestamp `26`. After that, the start signal `Signal 1` is requested at timestamp `27` and finally set at timestamp `34`. The approach-

ing train is simulated by setting the the sensors of the route to `HIGH`, respectively `LOW` in valid time intervals, beginning at timestamp `49`. Note that the test suite is designed to fulfill the needs of the standard. Coverage criteria may require additional test cases.

```
[0] Set Route request 1
[1] Assert Point 4 in state STRAIGHT
[26] Set Point 4 to STRAIGHT
[27] Assert Signal 1 in state GO, STRAIGHT, -1
[34] Set Signal 1 to GO, STRAIGHT, -1
[49] Set Sensor 14 to HIGH
[74] Set Sensor 14 to LOW
[89] Set Sensor 1 to HIGH
...
```

Mutant Testing To prove that the generated test suite is applicable and able to detect errors as demanded by CENELEC during validation, controller mutants are generated. These seed errors in the controller code that have to be found by some test in the generated test suite. Error seeding is based on knowledge of the system, i.e. on the TSTS that defines the dynamic semantics. Typical errors are e.g. missing transitions or missing, incorrect, or additional guards of transitions. These can be easily generated by changing operators or operands or skip transitions.

Simulation For each model, a simulation environment is automatically generated. The output is a Java program that uses the MetaEdit+ API and visualized state changes in the model. The environment is usable for three purposes: creating own tests by manual simulation, running the generated test suite on model level, and running the generated test suite for mutants to demonstrate the applicability of the generated test suite. At each point in time in a manual simulation, the model can be investigated as all variables values that are changed in the simulation are also set in the model. Consequently, the validation process can be run at each point in time to check that all requirements are fulfilled at runtime.

6 Conclusion

We have shown that a DSL for railway control systems can be designed by performing a domain analysis and realize it in a DSL framework. The language is easily usable by domain experts as the domain-specific notation is integrated. As the used framework MetaEdit+ generates the DSL editor automatically, the language can be directly used and tested. The degree of automation is higher than for GPLs as the DSL incorporates domain knowledge. Validation of models is automated as the relevant rules could be formalized in a checker. Testing is widely automated as relevant test cases can be automatically generated. The selection of relevant test data is based on the domain knowledge while their applicability is shown by mutant testing in a simulation environment. This environment enables also non-programmers to design test cases.

These results make it possible to design small, maintainable controllers for railway control systems. The high degree of automation in validation and verification reduces the development costs. Also, both tasks are optimized for the domain which improves the overall quality as relevant test cases are not forgotten. The approach can be further improved by including formal verification by bounded model checking as shown in [PBD+05]. Also, the possibility to model variants of the controller as described in [JOZ03] would increase the applicability of the approach in industry. Code generation is possible if interfaces for the available hardware are designed and made available in the model, e.g. in the specification diagram.

References

[Ben86] Jon Louis Bentley. Programming Pearls: Little Languages. *Commun. ACM*, 29(8):711–721, 1986.

[Ber07] Kirsten Berkenkötter. OCL-based Validation of a Railway Domain Profile. In Thomas Kühne, editor, *Models in Software Engineering*, number 4364 in Lecture Notes in Computer Science, pages 159–168, Berlin, Heidelberg, New York, 2007. Springer.

[BH06] Kirsten Berkenkötter and Ulrich Hannemann. Modeling the Railway Control Domain rigorously with a UML 2.0 Profile. In J. Górski, editor, *Safecomp 2006*, volume 4166 of *LNCS*, pages 398–411. Springer, 2006.

[CEN] Railway Applications - Communications, Signalling and Processing Systems - Software for Railway Control and Protection Systems. CENELEC, 2001. CENELEC EN 50128.

[CO02] Rachel Cardell-Oliver. Conformance Test Experiments for Distributed Real-Time Systems. In Ernst-Rdiger Olderog and Bernd Steffen, editors, *International Symposium on Software Testing and Analysis (ISSTA'02)*, volume 1710 of *ACM Press*, pages 159–163, 2002.

[Dot] Graphviz- Graph Visualization Software. http://www.graphviz.org/.

[GME] The Generic Modeling Environment. http://www.isis.vanderbilt.edu/projects/gme. Vanderbilt University, Nashville.

[JOZ03] Stan Jarzabek, Wai Chun Ong, and Hongyu Zhang. Handling variant requirements in domain modeling. *Journal of Systems and Software*, 68(3):171–182, 2003.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific Modeling*. John Wiley & Sons, 2008.

[Met] MetaEdit+. http://www.metacase.com. Metacase.

[Pac02] Joern Pachl. *Railway Operation and Control*. VTD Rail Publishing, Mountlake Terrace (USA), 2002. ISBN 0-9719915-1-0.

[PBD+05] Jan Peleska, Kirsten Berkenkötter, Rolf Drechsler, Daniel Große, Ulrich Hannemann, Anne E. Haxthausen, and Sebastian Kinder. Domain-Specific Formalisms and Model-Driven Development for Railway Control Systems. In *TRain workshop at SEFM2005*, September 2005.

[vDK98] Arie van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

Erfahrungen bei der modellbasierten Entwicklung von Fahrwerksregelfunktionen im AUTOSAR-Umfeld und notwendige Entwicklungsschritte

Karsten Schmidt
karsten.schmidt@audi.de

Philipp Janda
philipp.janda@ini.fau.de

Abstract: Die Nutzung modellbasierter Entwicklungsansätze ist im Automobilbereich bei der Funktionsentwicklung stark verbreitet. Dabei gilt es zu beachten, dass sich die Modelle, die in einem modellbasierten Entwicklungsprozess eingesetzt werden, an der Problem- statt an der Lösungsdomäne orientieren müssen. Die Nutzung von AUTOSAR als Entwicklungsansatz, um die Softwarekomplexität im Fahrzeug zu beherrschen, findet immer größere Verbreitung. Die Nutzung modellbasierter Ansätze im AUTOSAR-Umfeld gestattet neue Lösungsmöglichkeiten, erzeugt jedoch auch neue Herausforderungen. Die dabei notwendige Umstellung der Entwicklungsprozesse auf AUTOSAR eröffnet ideale Möglichkeiten, die E/E-Systementwicklung an den für den Kunden erlebbaren Funktionen auszurichten und Architekturaspekte frühzeitig zu berücksichtigen. Der vorliegende Aufsatz fasst den aktuellen Stand bei der Integration beider Ansätze zusammen und zeigt notwendige Aktivitäten auf.

1 Stand der Technik in der Steuergeräteentwicklung für Fahrzeuge

Die intelligente Nutzung der Potenziale von Software im Automobil erfolgt mit dem Ziel, dem Kunden ein Plus an Sicherheit, Komfort und Zuverlässigkeit zu bieten. Die damit einhergehende größere Funktionsvielfalt resultiert aber gleichzeitig in komplexeren Systemen. In einem Fahrzeug der Oberklasse arbeiten heute rund 70 vernetzte Steuergeräte, die typischerweise von unterschiedlichen Herstellern stammen [Rei09]. Die Integration dieser Steuergeräte im Gesamtfahrzeug stellt eine große Herausforderung für die Fahrzeughersteller dar.

Speziell im Bereich der Fahrwerkregelsysteme sind in den nächsten Jahren neue Funktionen zu erwarten. Weiterhin sind die Fahrzeughersteller bei der Integration dieser neuen Funktionalitäten mit einer explodierenden Vielfalt an Varianten und Modifikationen konfrontiert. Diese ergeben sich aus der baureihenübergreifenden Wiederverwendung von Komponenten, Evolutionsstufen im Laufe eines Fahrzeuglebenszyklus sowie – vor allem – den möglichen Ausstattungsvarianten. Die Vergabe der Entwicklungsaufträge an unterschiedliche Zulieferer erhöht den Aufwand, der zur Beherrschung dieser Vielfalt betrieben werden muss.

Diese enorme Komplexität, sowohl auf der Funktionsseite, der softwaretechnischen Umsetzung, als auch der zugrundeliegenden Vernetzung erfordert neue Ansätze während der Entwicklung. Aktuell sind mehrere Lösungswege erkennbar.

Zunächst ist festzustellen, dass die Nutzung modellbasierter Ansätze zur Umsetzung der eigentlichen Fahrzeugfunktion stark verbreitet ist. Tools wie Simulink, TargetLink oder ASCET sind weit verbreitete Werkzeuge. Diese ermöglichen sowohl die Formulierung von Ideen als auch deren Umsetzungen in einer Domäne (Sprache), die dem Funktionsentwickler, der typischerweise kein Softwareentwickler ist, vertraut ist.

Um die Basisfunktionen eines Steuergerätes realisieren zu können, findet die Nutzung des AUTOSAR-Standards eine immer größere Verbreitung. Die OEM versuchen damit die wachsende und problematisch werdende Komplexität der E/E-Systeme im Automobil beherrschbar zu halten [Sch08, HKL07, GGRS07]. Der Fokus liegt dabei auf der Trennung von Anwendungs- und Basissoftware, diese entsprechen der Implementierung von Funktion und Infrastruktur.

Der AUTOSAR-Standard bietet neben einer standardisierten und werkzeuggestützten Konfiguration der Hardware auch die Möglichkeit, Anwendungssoftware in Softwarekomponenten zu strukturieren und diese mit Hilfe einer Middleware transparent auch über Steuergerätegrenzen miteinander kommunizieren zu lassen. Richtig angewendet führt dieses Prinzip zu einer erhöhten Wiederverwendbarkeit und Portierbarkeit der Anwendungssoftware und damit zu einer Beschleunigung der Entwicklung zukünftiger Anwendungen, insbesondere in Produktlinien.

2 AUTOSAR

Zentrales Element des AUTOSAR-Konzepts ist die Entkopplung applikativer Logik von zugrunde liegender Hardware. Das Prinzip des virtuellen funktionalen Busses (Virtual Functional Bus, VFB) stellt dabei eine übergreifende Abstraktionsschicht dar, die als alleinige Entwurfsschnittstelle für die Anwendung hinsichtlich Systemaufrufe, Kommunikation und Ablaufplanung fungiert.

Auf Ebene eines Steuergeräts wird die Abstraktionsschicht von der Laufzeitumgebung (Run-Time-Environment, RTE) implementiert. Diese realisiert eine einfache und statisch konfigurierte Middleware für das Steuergerät. Zweck der RTE ist es, die hardwarenahen Schichten der Software vor den Anwendungen zu verbergen. Durch Verwendung dieser Abstraktionsschicht erreicht man eine Verschiebbarkeit zwischen Steuergeräten unterschiedlicher Hardwarearchitektur. Dieses Prinzip wird in Abbildung 1 verdeutlicht. Die Zuordnung der Anwendungen 1 und 2 auf Steuergeräte kann getauscht werden.

Die Kommunikation zwischen Anwendungen erfolgt unter Nutzung so genannter Kommunikationsports ohne Kenntnis des realen Signalpfades. Somit besteht die Möglichkeit, fahrzeugspezifische Funktionen zunächst ohne Kenntnis der im Zielsystem verwendeten Bustopologie zu entwickeln. Im späteren Entwicklungsablauf werden die tatsächlichen Signalpfade durch eine Zuordnung auf die topologiespezifische Konfiguration festgelegt. Die Anwendungssoftware eines Steuergeräts ist in unabhängigen Einheiten, den so genannten

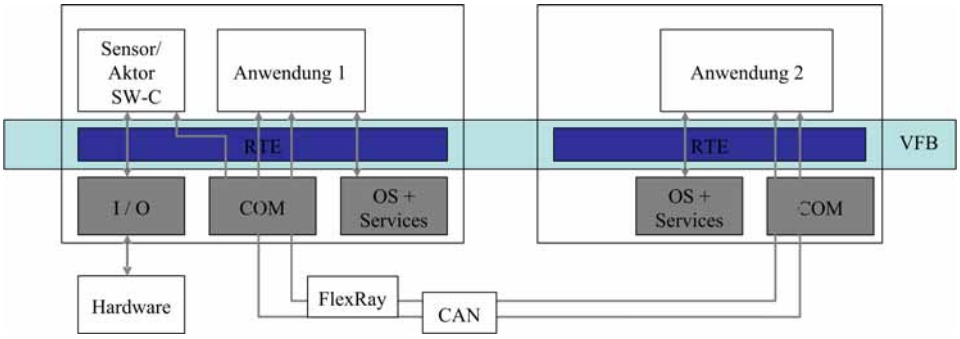


Abbildung 1: Austauschbarkeit von Softwarekomponenten

Softwarekomponenten (Software Components, SW-C), organisiert. Diese stellen das wichtigste Element zur Strukturierung der Softwarearchitektur dar. Eine solche Komponente versteckt die Implementierungsdetails und stellt definierte Schnittstellen zur Verfügung. Die Softwarekomponente kann als logischer Container aufgefasst werden, während die Runnable Entities den ausführbaren Code enthalten. Sie werden konfigurierten Tasks des Betriebssystems zugeordnet und von diesem entsprechend der gewählten Ablaufplanung aufgerufen.

3 Modellierung zeitlicher Zusammenhänge in Fahrzeugsystemen

Moderne mechatronische Systeme im Kraftfahrzeug werden immer häufiger als verteilte Regelungen realisiert. So können Sensoren, Steuergeräte und Aktoren räumlich im Fahrzeug verteilt sein und Messwerte sowie Stellbefehle werden über Bussysteme übertragen. Ein typisches Beispiel in modernen Fahrzeugen stellen Systeme mit so genannten intelligenten Stellern dar, in welchen ein Teil der Regelaufgabe von einem zentralen Steuergerät übernommen wird, andere Teile der Regelaufgabe dagegen dezentral in einem aktornahen Steuergerät ausgeführt werden [RSG+08, SRB+08]. Durch derartige Vernetzungen einzelner Regelsysteme zu einem komplexen, verteilten Regelsystem lässt sich in vielen Fällen eine deutliche Vergrößerung des Nutzens für den Fahrer erzielen. So kann eine fahrzeugweite Verwendung der Messwerte aller Sensoren erfolgen, und Zusatzfunktionalitäten können generiert werden [KDRS06, SKV+08]. Jedoch können durch die Funktionsverteilung auch unterschiedliche negative Effekte entstehen, wobei der offensichtlichste der entstehende Laufzeiteffekt ist, der entsprechend beim Systementwurf berücksichtigt und minimiert werden muss.

Beim Entwurf verteilter Funktionen ergibt sich die Notwendigkeit, Aspekte der Funktionsverteilung gemeinsam mit der Softwarearchitektur zu berücksichtigen, um die notwendigen weiteren Randbedingungen (z.B. die zu erfüllenden Zeitbedingungen) zu berücksichtigen. Dies erfordert einerseits die Bereitstellung eines entsprechenden Modells,

um die zeitlichen Systemeigenschaften entsprechend erfassen und modellieren zu können. Ein solches Modell muss in der Lage sein, die unterschiedlichen Anwendungsfälle im Entwicklungsprozess zu berücksichtigten.

Die Integration verschiedener – prozessbedingter wie technisch notwendiger – Randbedingungen in ein solches Modell ermöglicht es dabei, relevante Parameter für die Erfüllung zeitlicher Anforderungen (FlexRay-Zyklen, Latenzen, etc.) automatisiert durch Tools verarbeiten zu lassen [RSG+07].

Die Notwendigkeit einer exakten Systematik für die Behandlung zeitlicher Aspekte haben verschiedene Hersteller und Zulieferer erkannt, und im Rahmen dieses Prozesses wurde im Jahre 2006 das internationale Forschungsprojekt TIMMO [TIM] initiiert, das neben einer standardisierten Behandlung zeitlicher Randbedingungen während der Entwicklung auch eine frühe Analysierbarkeit und vereinfachte Verifikation als Ziele verfolgt. Ebenso wurde 2007 eine Arbeitsgruppe der AUTOSAR-Entwicklungspartnerschaft ins Leben gerufen, die sich eine ähnliche Aufgabe gestellt hat. Ziel der Bestrebungen ist es, in AUTOSAR Release 4.0 einen ersten Entwurf eines Timing-Modells zu integrieren, der für folgende Versionen mit Hilfe der TIMMO-Ergebnisse verfeinert werden kann.

4 Integration der Funktionsentwicklung in den Softwareentwicklungsprozess

Im Automotive-Bereich ist es üblich, Software auf hohem Abstraktionsniveau durch modellbasierte Entwicklung zu erstellen. Für viele Aufgaben stehen etablierte modellbasierte Werkzeuge und Codegeneratoren zur Verfügung, wie z.B. Simulink/Targetlink für die Entwicklung der entsprechenden Anwendungen.

Die Entwicklungsprozesse sind mittlerweile allerdings so komplex, dass sie nicht mehr nur durch ein modellbasiertes Werkzeug abgedeckt werden können und auch nicht mehr nur von einer Person bzw. Abteilung durchgeführt werden – die Integration der Erzeugnisse von mehreren modellbasierten Werkzeugen gewinnt damit an Bedeutung.

4.1 Entwicklungsschritte und Toolunterstützung

Im Folgenden sollen die Schritte einer Steuergeräteentwicklung dargestellt werden. Diese Schritte sind in Abbildung 2 abgebildet.

Bei der Reglerentwicklung von Fahrdynamikregelsystemen für ein AUTOSAR-basiertes Steuergerät kann der Fahrdynamikregler selbst beispielsweise mit Matlab/Simulink erstellt und via Targetlink in C-Code überführt werden. Ergebnis dieses Prozesses ist eine AUTOSAR-Komponentenbeschreibung im XML-Format und Quelltext, der bereits AUTOSAR-konforme Schnittstellenaufrufe beinhaltet.

Anschließend muss eine Software-Architektur erstellt werden, die die erstellte Softwarekomponente mit anderen Bausteinen wie weiteren Anwendungen sowie den

Basissoftware-Diensten verbindet. Für diesen Schritt bieten sich Werkzeuge wie SystemDesk der Firma dSPACE an.

Diese Software-Architektur geht wiederum in ein Konfigurationswerkzeug wie z.B. Tresos Studio ein. Dieses ist für die Erstellung des Betriebssystems sowie der durchgängigen Konfiguration der Basissoftware (bspw. Kommunikationsstack, HW-Treiber, ...) zuständig. Weiterhin gehen auch Daten aus anderen Informationsquellen, wie z.B. die Buskommunikation aus DBC-Datenbanken oder der Task-Schedule, in die Konfiguration ein. Diese Integration von Informationen aus verschiedenen Quellen und der Erzeugnisse der Entwicklungswerkzeuge wird derzeit noch nicht ausreichend unterstützt und läuft heutzutage weitgehend manuell ab. Die unterschiedlichen Tools erfordern jedoch jeweils entsprechendes Expertenwissen. Ziel des Projekts "Modellgetriebene Komponentenkomposition" ist die einfache Integration der verschiedenen Erzeugnisse modellbasierter Entwicklungswerkzeuge am Beispiel einer Fahrdynamikreglerentwicklung. Dabei ist es wichtig, dass der Funktionsentwickler alle notwendigen Schritte durchführen kann, um von einer Funktionsidee zu einem Steuergerät zu kommen.

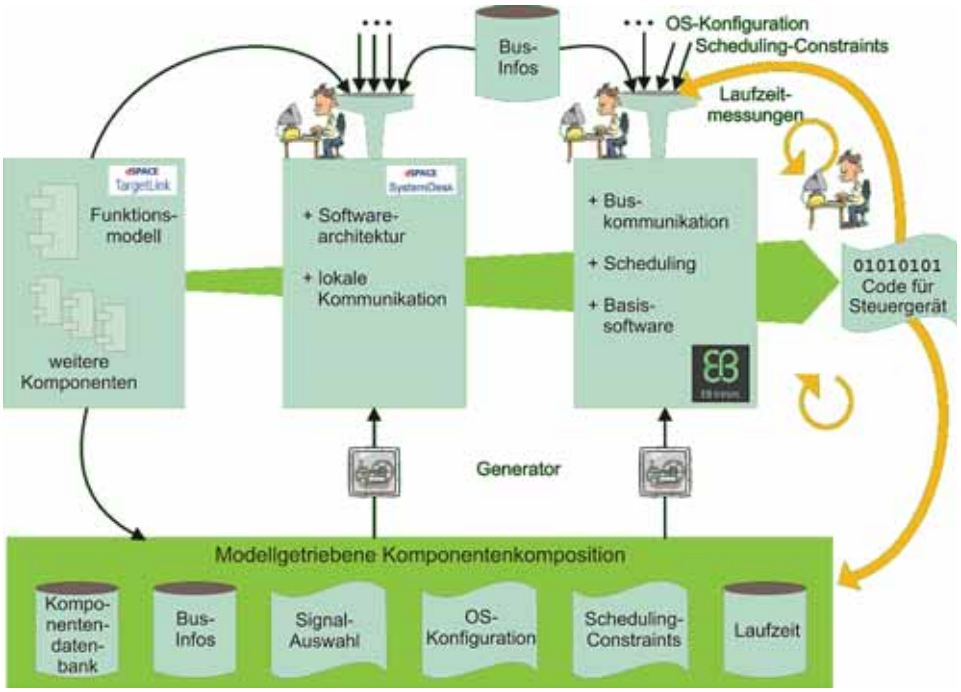


Abbildung 2: Aktueller Workflow (oben), Automatisierter Workflow (unten)

4.2 Schnittstellendatenbank

Ein Ziel des AUTOSAR-Standards ist die Wiederverwendung selbstgeschriebener Software und die Verwendung sog. COTS-Software (Commercial Off The Shelf). Ermöglicht wird dies durch die Strukturierung der Software in Komponenten sowie durch Verstecken der Implementierungsdetails hinter wohldefinierten und typsicheren Schnittstellen. Da die Einigung mehrerer Parteien auf gemeinsame Interfaces schwierig ist, definiert AUTOSAR die Kompatibilität von Interfaces.

Bereits innerhalb der unterschiedlichen Entwicklungsbereiche beim OEM existieren unterschiedliche Festlegungen über die verwendeten Schnittstellen, die sich zusätzlich noch zwischen den Fachbereichen unterscheiden können. Eine herstellerweite Festlegung von kompatiblen und zukunftssicheren AUTOSAR-Interfaces stellt somit eine große Herausforderung dar. Wünschenswert ist natürlich auch eine Abstimmung mit den Zulieferern.

Zentrale Bedeutung für die Kompatibilität von AUTOSAR-Schnittstellen haben die Datenelemente, insbesondere deren Namen, da sie für die Zuordnung der Daten der verschiedenen Interfaces benötigt werden. Um eine sinnvolle Definition von neuen Interfaces vornehmen zu können, ist daher eine Suche nach bestehenden Datenelementen aus bisherigen Fahrzeugprojekten (über unterschiedliche Suchkriterien) unumgänglich. Ein Datenelementkatalog, z.B. auf Basis einer relationalen Datenbank, bietet die geforderte Funktionalität und unterstützt somit den Entwicklungsprozess.

4.3 Aspektorientierung

Die AUTOSAR-Basissoftware ist streng in Module gegliedert, um einerseits eine Modellierung eines AUTOSAR-Steuergerätes zu ermöglichen, andererseits die Kombination verschiedener Teile der Basissoftware von unterschiedlichen Herstellern zu gestatten. Unerwünschter Nebeneffekt dieser Trennung ist jedoch, dass eine globale Optimierung der Basissoftware in Bezug auf übergreifende Kriterien wie z.B. Speicherbedarf, Energieverbrauch, etc. erschwert wird. Parallel zur Theorie der Programmiersprachen handelt es sich dabei um sog. querschneidende Belange, denen man mit Aspektorientierter Programmierung begegnen kann. Aspekte bündeln dabei über mehrere Module oder Klassen verteilte Funktionalitäten und fügen die nötigen Änderungen in einem separaten Generierungsschritt in den Quellcode ein.

Hier sind die Toolhersteller gefordert, ihre Werkzeuge um entsprechende Möglichkeiten zu erweitern und eine modulübergreifende Optimierung der Software zu ermöglichen. Der AUTOSAR-Standard bietet bereits mit den Konformitätsklassen (Implementation Conformance Classes, ICC) die Möglichkeit Optimierungen durchzuführen und so den Anforderungen an die entsprechende Zielhardware zu genügen. Jedoch geht der hier beschriebene Wunsch über die reinen Konformitätsklassen hinaus, da die Modellierung des AUTOSAR-Systems komplett über alle Schichten erfolgen muss, jedoch die Optimierung nach unterschiedlichen Aspekten erfolgt.

5 Anforderungen an die weitere AUTOSAR-Entwicklung

Der AUTOSAR-Standard hat mit der Version 3.0 einen stabilen Stand erreicht, der es ermöglicht, Serienprojekte auf breiter Front zu starten. Um die Akzeptanz auf Seiten der OEM, Tier-1 und Toolhersteller zu stabilisieren, ist es erforderlich, eine herstellerübergreifende Abstimmung über den zu verwendenden Standard zu etablieren. Diese ermöglicht dann den Toolherstellern, stabile und breit akzeptierte Werkzeuge anzubieten.

Jedoch ist auch die kontinuierliche Weiterentwicklung des Standards notwendig. Wünschenswert wäre eine Integration des Timingmodells in den AUTOSAR-Standard, wie bereits in [SR08] diskutiert. Des Weiteren ist es es sinnvoll, dass die Sicherheitsanforderungen der IEC 61508 [IEC] und der ISO 26262 Berücksichtigung bei der Weiterentwicklung der Entwicklungswerkzeuge und des AUTOSAR-Standard finden.

Die modellbasierte Entwicklung speziell im AUTOSAR-Umfeld ermöglicht und erfordert neue Wege der Zusammenarbeit sowohl innerhalb der Automobilhersteller als auch zwischen den Zulieferern und OEM. Die notwendigen Prozessanpassungen erfordern eine entsprechende Prozessmodellierung. Dazu bietet sich SPEM an. Erste Untersuchungen sind bereits erfolgt [BJKS08].

Ein interessanter Aspekt bei einer durchgängigen Nutzung der Möglichkeiten, die AUTOSAR bietet, stellt die virtuelle Integration von Reglerfunktionalitäten dar [SG08]. Die konsequente Nutzung der Möglichkeiten, die eine virtuelle Integration bietet, eröffnet ein großes Potential zur Kosteneinsparung.

Literatur

[BJKS08] Max Brunner, Martin Jung, Detlef Kips und Karsten Schmidt. Fallstudie zur Modellierung von Software-Entwicklungsprozessen auf Basis von SPEM 2.0. In *Software Engineering 2008*, München, 2008. TU München.

[GGRS07] Frank Grosshauser, Frank Gesele, Stephan Reichelt und Karsten Schmidt. In die Realität überführt; Nutzung von AUTOSAR in der Serie. *elektronik automotive*, Sonderausgaube AUTOSAR:36–40, 2007. ISSN: 1614-0125.

[HKL07] Hubert Hietl, Jens Kötz und Günter Linn. Bereit für FlexRay. *Elektronik automotive*, Seiten 32–36, 2007.

[IEC] IEC 61508. *Functional safety of E/E/PE safety-related systems*. Norm.

[KDRS06] H. Krimmel, H. Deiss, W. Runge und H. Schürr. Elektronische Vernetzung von Antriebsstrang und Fahrwerk. *ATZ*, 5, 2006.

[Rei09] Konrad Reif. *Automobilelektronik*. ATZ/MTZ-Fachbuch. Vieweg-Teuber, Wiesbaden, 3. Auflage, 2009.

[RSG+07] Stephan Reichelt, Karsten Schmidt, Frank Gesele, Nils Seidler und Wolfram Hardt. Nutzung von FlexRay als zeitgesteuertes automobiles Bussystem im AUTOSAR-Umfeld. In Peter Holleczek und Birgit Vogel-Heuser, Hrsg., *Mobilität und Echtzeit*, Seiten 79–87. Gesellschaft für Informatik, Springer, 2007.

[RSG+08] Konrad Reif, Karsten Schmidt, Frank Gesele, Stephan Reichelt, Martin Saeger und Nils Seidler. Vernetze regelsysteme im Kraftfahrzeug. *ATZelektronik*, 4:32–38, 2008.

[Sch08] M. Schöttle. AUTOSAR sorgt für einen Paukenschlag, 2008. Interview mit Elmar Frickenstein (BMW).

[SG08] Karsten Schmidt und Frank Gesele. Systematisches Testkonzept für AUTOSAR-basierte Softwarekomponenten. In *Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik*, Berlin, 2008.

[SKV+08] H. Snakman, P. Köhn, H. Vieler, M. Krenn und D. Odenthal. Integrated Chassis Management ein Ansatz zur Strukturierung der Fahrdynamikregelsysteme. In *17. Aachener Kolloquium Fahrzeug- und Motorentechnik*, 2008.

[SR08] O. Scheickl und M. Rudorfer. Automotive Real Time Development Using a Timing-augmented AUTOSAR Specification. In *ERTS2008*, 2008.

[SRB+08] Karsten Schmidt, Stephan Reichelt, Andreas Baudisch, Frank Gesele, Nils Seidler, Martin Saeger und Konrad Reif. Entwurfaspekte verteilter Regelsysteme im Kraftfahrzeug. In *17. Aachener Kolloquium für Fahrzeug- und Motorentechnik*, Seiten 1721–1741, Aachen, 2008.

[TIM] TIMMO. Timing Model. www.timmo.org.

Feature-basierte Modellierung und Verarbeitung von Produktlinien am Beispiel eingebetteter Software

[1]Christian Berger, [2]Holger Krahn, [1]Holger Rendel, [1]Bernhard Rumpe

[1]RWTH Aachen
Lehrstuhl für Software Engineering
Ahornstraße 55
52074 Aachen
http://www.se-rwth.de

[2]TU Braunschweig
Institut für Software Systems Engineering
Mühlenpfordtstraße 23
38106 Braunschweig
http://www.sse-tubs.de

Abstract: Verwandte Baureihen elektronischer Geräte haben typischerweise bezüglich der eingebetteten Software große Ähnlichkeit zueinander. Durch die Etablierung einer Software-Produktlinie lassen sich sowohl Kosten reduzieren als auch die allgemeine Qualität steigern. In dieser Arbeit wird eine Sprache zur Modellierung und Verarbeitung von Software- und Systemvarianten vorgestellt, die modular durch einbettbare Nebenbedingungen zur Prüfung semantischer Korrektheit über die Semantik der bekannten Feature-Diagramme hinaus erweitert werden kann. Diese Modellierung der Variabilität wird durch etablierte Funktionsnetze zur Beschreibung der logischen Architektur ergänzt.

1 Einleitung und Motivation

Eingebettete Software zur Steuerung elektronischer Systeme findet sich in vielen Gegenständen des Alltags. Häufig bietet es sich an, dass Software für die Varianten eines elektronischen Geräts in ihren Ausbaustufen und ihren über die Jahre optimierten Baureihen wiederverwendet wird, da die darin enthaltenen Steuerungskomponenten häufig große Ähnlichkeiten zueinander aufweisen. Das gilt für Drucker genau so wie für Fernseher oder Steuergeräte im Flugzeug oder Auto. Aufgrund wachsenden Kundenanspruchs steigt aber die Komplexität der Software und damit der Aufwand in Wartung und Weiterentwicklung.

Weisen Systemkomponenten und insbesondere Software-basierte Teilkomponenten nach Durchführung einer Domänenanalyse [KCH+90] große Übereinstimmungen und damit eine hohe Ähnlichkeit auf, bietet sich die Einführung einer Software-Produktlinie an [CN02]. Software-Produktlinien erlauben hohe Einsparpotenziale, insbesondere in der Wartung und Pflege von Software-Artefakten und ermöglichen eine strukturierte Wiederverwendung der Software für die identifizierten Varianten [PBL05].

In dieser Arbeit wird die Modellierung von System- und insbesondere Software-Komponenten durch spezielle domänenspezifische Sprachen (engl. domain specific languages – DSLs [DKV00, MHS05, GKR+07]) vorgestellt, die um flexible Regelwerke modular ergänzt werden kann. Die Anwendung und Verarbeitung der Feature-DSL wird am Beispiel eines Navigationssystems vorgestellt und ihr Potenzial im Rahmen eines Software-Entwicklungsprozesses diskutiert, der auf Funktionsnetzen basiert.

2 Feature-basierte Modellierungssprache und Verarbeitung

Die in Abbildung 1 dargestellte Feature-DSL enthält sämtliche syntaktischen Elemente eines Feature-Diagramms nach [CE00] in textueller und somit maschinenverarbeitbarer Form. Die entworfene DSL und das dazugehörige Werkzeug basieren auf dem MontiCore-Framework [GKR+06, KRV07, GKR+08, KRV08].

```
01 external Constraint;
02
03 FeatureDiagram = "featurediagram" name:IDENT "{"
04   head:FeatureInTree
05   FeatureInTree*
06   "constraints" "{"
07     (Constraint ";")*
08   "}"
09 "}";
10
11 Feature = name:IDENT (optional:["?"])?;
12
13 interface FeatureInTree;
14
15 AllFeature implements FeatureInTree = name:IDENT "="
16   Feature ("," Feature)* ";";
17
18 AlternativeFeature implements FeatureInTree = name:IDENT "="
19   "alt" "(" Feature ("," Feature)* ")" ";";
20
21 OrFeature implements FeatureInTree = name:IDENT "=" "or" "(" Feature
22   ("," Feature)* ")" ";";
23
24 MultipleFeature implements FeatureInTree = name:IDENT "="
27   "[" lowerbound:INT ".." upperbound:INT "]"
28   "(" Feature ("," Feature)* ")" ";";
```

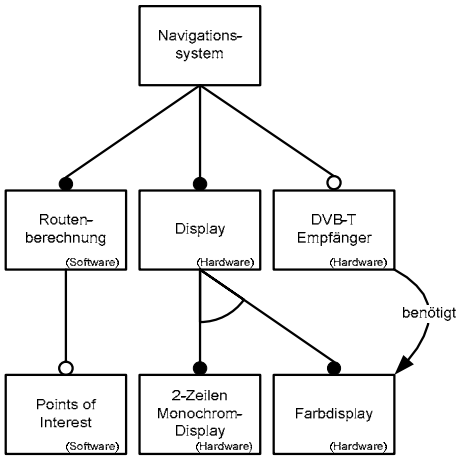
Abbildung 1: Auszug aus der Grammatik der Feature-DSL.

Die zentrale Regel der Grammatik ist in den Zeilen 3-9 zu finden, in denen eine Beschreibung für Feature-Diagramme durch das Schlüsselwort „featurediagram“ mit einem Namen eingeleitet wird. Anschließend folgen in einem Block, der in geschweiften Klammern gefasst ist, die einzelnen Features, von denen das erste den Kopf des Diagramms bildet (Zeile 4). Von der Grammatik werden folgende Features unterstützt: Atomare Features (Zeile 11), aus mehreren benötigten Subfeatures bestehende Features (Zeile 15), alternativ ausschließende Features (Zeile 18), Or-Features, bei denen mehrere Subfeatures ausgewählt werden können (Zeile 21) und die Möglichkeit aus einer Anzahl an gegebenen Features eine bestimmte Menge zu selektieren (Zeile 24).

Das besondere Merkmal hierbei sind modular einbettbare Subsprachen, hier konkret einer Logik-Sprache, zur Definition von Nebenbedingungen (Zeile 1 und 7). Durch das nicht verfeinerte Nichtterminal „Constraint“ in der obigen Grammatik wird eine Austauschbarkeit und zudem eine Erweiterbarkeit der Feature-Sprache gewährleistet, so dass die oben definierte Grammatik und die darauf aufbauenden Werkzeuge wiederverwendet werden können. In den folgenden Beispielen wird für „Constraint“ eine einfache Aussagenlogik verwendet.

3 Feature-basierte Modellierung am Beispiel eines Navigationssystems

Als ein einfaches System, in dem die vorgestellten DSLs eingesetzt werden, wird ein Navigationssystem modelliert. Dieses System ist in grafischer und textueller Form in Abbildung 2 dargestellt.




```
01 featurediagram Navi {
02
03   Navigationssystem =
04     Routenberechnung,
05     Display, DVBTEmpfaenger;
06
07   Routenberechnung =
08     PointsofInterest?;
09
10   Display = alt
11    (Monochromdisplay,
12    Farbdisplay);
13
14   constraints {
15     DVBTEmpfaenger ->
16       Farbdisplay;
17   }
18 }
```

Abbildung 2: Modellierung eines Navigationssystems als Feature-Diagramm.

Das dargestellte Beispiel beschreibt ein vereinfachtes Navigationssystem, von dem vom Diagramm die Teile der Routenberechnung, das Display und ein optionaler DVB-T Empfänger berücksichtigt werden (Zeile 3-5). Die Routenberechnung kann optional durch eine Points of Interest-Datenbank erweitert werden (Zeile 7-8). Beim Display steht alternativ eine Monochrom-Text- oder Farbanzeige zur Auswahl (Zeile 10-12). Bei der Benutzung des DVB-T Empfängers wird letztere zwingend benötigt (Zeile 14-17). Dies ist eine Nebenbedingung, die in der oben erwähnten einfachen Aussagenlogik eingebettet wurde.

Falls es der Kontext erfordert, dass Nebenbedingungen angegeben werden müssen, für die eine einfache Aussagenlogik nicht ausreicht, kann ein anderes Regelwerk verwendet werden. Beispielsweise könnte eine Points of Interest-Datenbank zur Intellectual Property (IP) eines Kunden gehören. Durch die Definition von Constraints in der zur Verfügung stehenden Regelwerk-Sprache kann festgelegt werden, dass diese Komponente nur Produktvarianten dieses Kunden zur Verfügung steht. Dafür ist aber eine komplexeres Regelwerk als die zunächst eingebettete Aussagenlogik nötig, die zusätzlich auf elektronisch verfügbare Vertragswerke zugreift.

Als Erweiterung kann ein Standard- und eine Premium-System des Navigationssystems angegeben werden. Die Beschreibung der Systeme, wie sie im „constraints“-Block auftauchen würde, ist in Abbildung 3 dargestellt.

```
[..]
01   Standard -> (Routenberechnung & Monochromdisplay);
02   Premium -> (Routenberechnung & PointsofInterest & Farbdisplay &
       DVBTEmpfaenger);
[..]
```

Abbildung 3: Definition von Systemen.

4 Funktionsnetze

Eine praktische Anwendung der Modellierungssprache ergibt sich im Zusammenhang mit Funktionsnetzen durch Sichtenbildung, wie sie in [GHK+08a, GHK+08b] beschrieben werden. Sie stellen eine Möglichkeit zur Modellierung der logischen Architektur eines Softwaresystems dar. In [GKPR08] wird beschrieben, wie der Sichtenmechanismus dazu verwendet werden kann, Varianten innerhalb einer automotiven Baureihe zu modellieren. Hier soll nun aufgezeigt werden, welche Ergänzungen zur Etablierung einer Software-Produktlinie notwenig sind.

Der bisherige Ansatz erlaubt nicht die explizite Formulierung und Überprüfung von Nebenbedingungen. Die modulare Erweiterung der Feature-DSL ergänzt daher eine einfache Aussagenlogik.

Bei der Anwendung der Funktionsnetze für eine Produktlinie kann diese als 150%-Modell modelliert werden. Dieses 150%-Modell enthält alle Teile, aus denen ein mögliches Produkt bestehen kann, das folglich einen Ausschnitt dieses Modells darstellt. Die einzelnen Features entsprechen ein oder mehreren Blöcken des Funktionsnetzes. Eine Konfiguration des Feature-Diagramms entspricht einer bestimmten Sicht auf das Funktionsnetz. So werden dann die einzelnen Produkte als Sichten auf dieses Gesamtmodell beschrieben und enthalten nur die für die Variante relevanten Blöcke.

Bei der Verwendung für Software-Produktlinien zeigt sich jedoch, dass bei neuen Varianten gezielt Elemente vom 150%-Modell abweichen sollen und daher nicht über den bisherigen Sichtenmechanismus beschrieben werden können. Daher muss die vorhandene Notation so erweitert werden, dass diese Differenzen beschrieben werden können und in die vorhandene Methodik einbettbar sind. Diese Differenzen werden basierend auf einer Referenzarchitektur angegeben, wie sie bei Software-Produktlinien verwendet wird. An diesem Punkt werden wir in Zukunft arbeiten.

5 Verwandte Arbeiten

Eine Grammatik für Feature-Diagramme ist in [DK01] zu finden. An einem Beispiel wird die Benutzung der Sprache verdeutlicht und ein Datenmodell für das Feature-Diagramm entworfen. Die Grammatik bietet jedoch weniger Möglichkeiten als die hier vorgestellte und ist an eine feste Definition von Nebenbedingungen gebunden.

[CE00] beschäftigt sich ebenfalls mit Feature-Diagrammen und deren Nebenbedingungen. Die hier vorgestellte DSL baut auf dieser Darstellung auf. In weiteren Veröffentlichungen werden Erweiterungen wie explizit definierbare Kardinalitäten zu den Diagrammen vorgestellt [CHE05].

In [PBL05] wird im Rahmen der Software-Produktlinien auf Variabilität eingegangen. Hier wird ein Variabilitätsmodell vorgestellt, das dem hier verwendeten ähnlich ist. Der größte Unterschied im Vergleich zu dem hier verwendeten Modell besteht darin, dass dieses explizit auf Variationspunkte und Varianten eingeht und nicht auf Features.

Ein Ansatz, Feature-Diagramme mittels UML-Klassendiagrammen darzustellen, wird in [Gom04] gezeigt. Sämtliche Nebenbedingungen sind im Diagramm angegeben und müssen nicht textuell festgehalten werden, was jedoch bei komplizierteren Konstellationen unübersichtlich werden kann. Austausch- und Erweiterungsmöglichkeiten bestehen um Rahmen der UML 2.0 [OMG07a, OMG07b].

Auch in anderen Arbeiten werden Sichten benutzt, um mögliche Produkte innerhalb einer Produktlinie zu modellieren. So stellt das Fraunhofer ISST in [GKM07] das VEIA-Projekt vor, welches ebenfalls ein 150%-Modell nutzt, um mehrere mögliche Systeme zu beschreiben. Darauf aufbauend wird das System aXBench [aXB] entwickelt, das eine Modellierung mittels Eclipse [Ecl] anbietet. Hier wird aber nicht auf die Möglichkeit eingegangen, bestimmte Systeme mittels Feature-Diagrammen zu spezifizieren.

6 Zusammenfassung und Ausblick

In diesem Papier wurde eine textuelle, maschinenverarbeitbare Sprache zur Modellierung von Feature-Diagrammen vorgestellt. Die Besonderheit der entworfenen Sprache stellt die Parametrisierung der Feature-Sprache mit einer nicht weiter definierten Constraint-Sprache dar. Diese kann durch Einbettung einer entsprechenden Grammatik festgelegt werden und stellt damit gleichzeitig eine Erweiterung der Sprache und der Werkzeuge zur Sprachverarbeitung dar. Die Anwendbarkeit der erweiterbaren Feature-DSL wurde an einem Beispiel zur Modellierung eines Navigationssystems demonstriert.

Die dargestellte Feature-DSL kann zum Beispiel bei der Umstellung von eingebetteter Software zur Steuerung elektronischer Geräte auf eine Software-Produktlinie die durch eine Domänenanalyse identifizierten Software- und Systemvarianten modellieren. Dazu lassen sich Funktionsnetze zur Beschreibung der logischen Architektur der Produktlinien und Sichten zur Modellierung der konkreten Varianten einsetzen. Derzeit werden Möglichkeiten zur Erweiterung der Notation untersucht, um Abweichungen der Varianten von einer vorgegebenen Referenzarchitektur der Produktlinie beschreiben zu können, die die vorhandene Sichtenbildung ergänzen sollen.

Die Integration der modularen Feature-DSL in einen Software- und Systementwicklungsprozess wird derzeit im Rahmen der Einführung einer Software-Produktlinie bei einem Hersteller von Steuergeräten untersucht. Hierbei zeichnet sich die modular erweiterbare Feature-DSL als ein vielversprechender Weg ab, unter anderem vorhandene Versionierungssysteme und System-Architekturen zur Generierung funktionsfähiger Software zur Übertragung auf ein Steuergerät zu integrieren.

Literaturverzeichnis

[aXB] Fraunhofer-Institut für Software- und Systemtechnik: *aXBench*. http://axbench.isst.fraunhofer.de/, 2009-03-24.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming: Methods, Tools, Applications*. Addison-Wesley, 2000.

[CHE05] Czarnecki, K.; Helsen, S.; Eisenecker U. W.: *Formalizing Cardinality-based Feature Models and their Specialization*. In: Software Process Improvement and Practice, Special Issue of Best Papers from SPLC04, 2005.

[CN02] Clements, P.; Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston 2001.

[DKV00] van Deursen, A.; Klint, P., Visser, J.: *Domain-specific languages: an annotated bibliography*. *SIGPLAN Not., ACM,* 2000*, 35*, 26-36.

[DK01] van Deursen, A.; Klint, P.: *Domain Specific Language Design Requires Feature Descriptions*. In: Journal of Computing and Information Technology 2001.

[Ecl] Eclipse Foundation: *Eclipse*. http://www.eclipse.org/, 2009-03-24.

[GHK+08a] Grönniger, H.; Hartmann, J.; Krahn, H.; Kriebel, S.; Rothhardt, L.; Rumpe, B.: *Modelling Automotive Function Nets with Views for Features, Variants, and Modes*. In: 4th European Congress ERTS – Embedded Real Time Software (2008).

[GHK+08b] Grönniger H.; Hartmann J.; Krahn, H.; Kriebel, S.; Rothhardt, L.; Rumpe, B.: *View-Centric Modeling of Automotive Logical Architectures.* In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV. Informatik-Bericht 2008-02, CFG-Fakultät, TU Braunschweig, 2008.

[GKM07] Große-Rhode, M.; Kleinod, E.; Mann, S.: *Entscheidungsgrundlagen für die Entwicklung von Softwareproduktlinien*. Fraunhofer-Institut für Software- und Systemtechnik, Abt. Verlässliche technische Systeme, 2007, ISST-Bericht 83/07, 2007.

[GKR+07] Grönniger, H.; Krahn, H.; Rumpe, B.; Schindler, M.; Völkel, S.:*Textbased Modeling*. 4th International Workshop on Software Language Engineering, 2007.

[GKR+08] Grönniger, H.; Krahn, H.; Rumpe, B.; Schindler, M. & Völkel, S.: *MontiCore: A Framework for the Development of Textual Domain Specific Languages.* 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume, 2008, 925-926.

[GKPR08] Grönniger, H.; Krahn, H.; Pinkernell, C.; Rumpe, B.: *Modeling Variants of Automotive Systems using Views*. In: Tagungsband Modellierungs-Workshop MBEFF: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (2008).

[GKR+06] Grönniger, H.; Krahn, H.; Rumpe, B.; Schindler, M.; Völkel, S.: MontiCore 1.0 – *Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen.* Technischer Report, Informatik-Bericht 2006-04, Institut für Software Systems Engineering, TU Braunschweig, 2006.

[Gom04] Gomaa, H.: *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. Addison-Wesley Object Technology Series, 2004.

[KCH+90] Kang, K.; Cohen, S.; Hess, J.; Nowak, W.; Peterson, S.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 1990.

[KRV07] Krahn, H.; Rumpe, B.; Völkel, S.: *Integrated Definition of Abstract and Concrete Syntax for Textual Languages*. Proceedings of Models 2007, 2007, 286-300.

[KRV08] Krahn, H.; Rumpe, B.; Völkel, S.: *MontiCore: Modular Development of Textual Domain Specific Languages*. Proceedings of Tools Europe, 2008.

[MHS05] Mernik, M.; Heering, J., Sloane, A. M: *When and how to develop domain-specific languages*. ACM Comput. Surv., ACM Press, 2005, 37, 316-344.

[PBL05] Pohl, K; Böckle, G.; van der Linden, F.: *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[OMG07a] Object Modeling Group: *Unified Modeling Language: Superstructure Version 2.1.2*, November 2007.

[OMG07b] Object Modeling Group: *Unified Modeling Language: Infrastructure Version 2.0*, November 2007.

Automatische Analyse und Generierung von AUTOSAR - Konfigurationsdaten

Jan Meyer, Wilhelm Schäfer

Universität Paderborn
s-lab (Software Quality Lab)
Warburgerstr. 100
33098 Paderborn

jmeyer@s-lab.upb.de
wilhelm@upb.de

Abstract: Die Bedeutung des AUTOSAR Architektur-Standards nimmt in der Automobilindustrie zu. Der Standard bringt jedoch nicht nur standardisierte Schnittstellen mit sich. Vielmehr beinhaltet er auch eine eigene Entwicklungsmethodik. Diese beruht auf der Konfiguration einer konkreten Systemarchitektur auf der Basis des Standards, sowie der automatischen Generierung der Software. Es werden aber keine Aussagen darüber getroffen, wie die notwendigen Daten für die Konfiguration ermittelt werden können. In diesem Positionspapier wird eine werkzeuggestützte methodische Vorgehensweise skizziert, wie Daten für die Konfiguration zunächst identifiziert, durch ein Unified Modeling Language (UML)-Modell formal präzisiert und analysiert, sowie automatisch in ein AUTOSAR kompatibles Modell transformiert werden können.

1 Einleitung

Die Vielzahl von Innovationen im Automobilbereich, welche zu großen Anteilen durch Software realisiert werden, lässt die Komplexität des Gesamtsystems „Kraftfahrzeug“ bekanntermaßen ansteigen. Grund hierfür ist die hohe Anzahl von Steuergeräten (Embedded Control Units – ECUs), welche über heterogene Netzwerke miteinander kommunizieren und somit ein komplexes Netzwerk von Funktionen realisieren. Betrachtet man die einzelnen ECUs eines Fahrzeugs, so ist auch hier ein Zuwachs an Komplexität zu erkennen. Leistungsfähigere Hardware ermöglicht die Integration von Funktionen auf einer ECU, welche vormals auf mehrere ECUs verteilt waren. Einige Funktionen, z.B. Fahrerassistenzsysteme, welche auf einer Echtzeit-Bilddatenverarbeitung basieren, bedingen zudem den Einsatz mehrerer Rechenkerne (Mikrocontroller, DSPs oder FPGAs).

Ein wesentliches Ziel des AUTOSAR Standards und der damit verbundenen Entwicklungsmethodik ist es, die so entstandene (Software-) Komplexität beherrschbar zu machen. Grundlage hierfür ist eine standardisierte System-Architektur, welche u.a. die Entwicklung von AUTOSAR-Software Komponenten unterstützt, die flexibel auf unterschiedliche ECUs verteilt werden können. Dies wird durch standardisierte Schnittstellen, sowie eine standardisierte Laufzeitumgebung – den Virtual Functional Bus (VFB) und seine technische Realisierung – die Runtime Environment (RTE) ermöglicht.

Die AUTOSAR Entwicklungsmethodik basiert auf der Konfiguration einer konkreten Systemarchitektur entsprechend dem Standard, sowie einer anschließenden automatischen Generierung des Quellcodes. Weitere Details werden in Kapitel 2 erläutert. Die Entwicklungsmethodik lässt aber offen, wie die Daten für die Konfiguration der AUTOSAR-kompatiblen Architektur ermittelt und ggf. auf Korrektheit und Konsistenz überprüft werden. Es handelt sich um einen rein manuellen Prozess, der typischerweise „nur“ eine informale Anforderungsbeschreibung als Ausgangspunkt hat und dementsprechend fehleranfällig ist.

Die Konfigurationsdaten umfassen dabei verschiedene Bereiche. Sie beinhalten beispielsweise die Verteilung der Software auf die Hardware. Außerdem muss auch die verwendete Basissoftware konfiguriert werden. Hierunter ist die Speicherbelegung, die Verwendung von AUTOSAR Services (z.B. Timer) und auch die Konfiguration der Kommunikation (Zuordnung von Kommunikationssignalen) zu sehen. Weiterhin wird die zeitliche Performance des Systems maßgeblich durch die Definition von Tasks, die durch Komposition oder Zerlegung von gewünschten Funktionen des Systems entstehen, sowie deren Laufzeit und Priorität beeinflusst. Dieser Teil der Konfiguration kann aufgrund einer großen Zahl von Abhängigkeiten ein kompliziertes „Scheduling“ bedeuten, was manuell nur mit hohem Aufwand bei hoher Fehleranfälligkeit durchführbar ist. Beispielsweise regeln die Tasklaufzeiten die Aktivierung der zyklisch aufzurufenden Funktionen. Oftmals werden Daten von den Sensoren passend zur Aktivierung der Funktion bereitgestellt. Bei Änderungen an den Laufzeiten ist es möglich, dass nicht aktuelle Informationen genutzt werden. Von daher besteht eine Abhängigkeit zwischen der Aktivierung der Funktion und der Bereitstellung der Sensordaten. Eine werkzeugunterstützte methodische Vorgehensweise für die Erstellung von Konfigurationsdaten wie sie z.B. in [NWW+06] gefordert wird, ist deshalb unumgänglich und das Thema dieses Papiers.

2 Grundlagen

Die zur Konfiguration des AUTOSAR-Modells notwendigen Informationen sind größtenteils in einem typischerweise in Prosatext formulierten Anforderungsdokument zu finden, das funktionale und nicht funktionale Anforderungen beinhaltet [WTS08]. Aus dieser informellen Beschreibung werden manuell die AUTOSAR-Konfigurationsdaten in einem aufwändigen und fehleranfälligen Prozess durch den Systemanalysten und Systemarchitekten ermittelt. Der Prozess wird methodisch nicht weiter unterstützt. Es fehlt vor allem ein formales Modell, das eine Analyse der ermittelten Daten auf Korrektheit und Konsistenz unterstützt. Die möglicherweise entstehenden Fehler können heutzutage nur durch intensive Tests und Simulationen in späteren Entwicklungsphasen gefunden werden, was zu hohen Kosten führt. Von daher ist es ratsam bereits in den frühen Entwicklungsphasen valide und korrekte Modelle zu besitzen.

In diesem Positionspapier wird die Lücke zwischen den informalen Anforderungen und dem konfigurierten AUTOSAR Modell durch ein formales Analysemodell geschlossen. In dem Analysemodell werden die einzelnen Anforderungen in einer formalen Form modelliert und analysiert. Zur Darstellung der Analyseergebnisse wird eine Erweiterung der Unified Modeling Language (UML) verwendet. Das erstellte Modell wird nicht nur zu Dokumentationszwecken [KeS07] genutzt, beispielsweise für die Zeitbudgets, die in den Anforderungen beschrieben sind. Vielmehr werden die angegebenen Zeitbudgets auch ausgewertet und diese Ergebnisse dienen als Entscheidungsgrundlage für die Designphase. Beispielhaft werden in diesem Papier Daten für die Konfiguration des AUTOSAR Betriebssystems aus dem Analysemodell extrahiert und somit für die Konfiguration des AUTOSAR Modells genutzt. Dabei liegt der Schwerpunkt auf der Konfiguration des Betriebssystems mit der Erstellung von Betriebssystemtasks, ihrer Priorisierung und der Zuordnung von AUTOSAR Runnables zu den Tasks.

2.1 Der AUTOSAR Standard

Der AUTOSAR Standard [Auto08] wurde von Herstellern, Zulieferern und Toolherstellern entwickelt, um eine gemeinsame Architektur aller Software-Komponenten und deren Zusammenspiel in einem Automobil zu definieren. Er zeichnet sich durch standardisierte Schnittstellen, einen virtuellen Bus und einer eigenen Methodik aus. Damit werden einzelne Komponenten leichter austauschbar und wieder verwendbar.

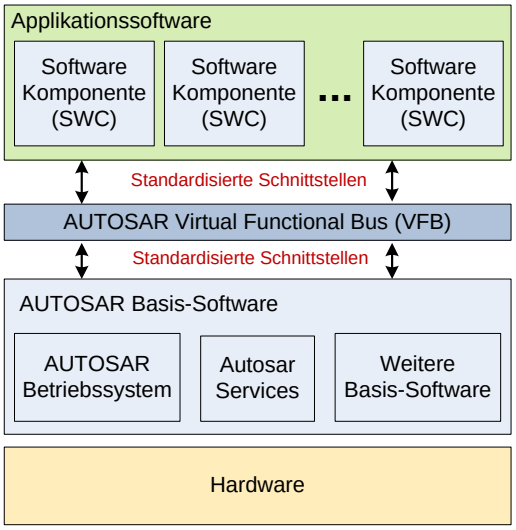


Abbildung 1: Architektur einer AUTOSAR Komponente

Mittels der Schnittstellen und einer Abstraktion durch verschiedene Schichten (vgl. Abbildung 1) ist es möglich Applikationen auf verschiedene Steuergeräte zu verteilen. Eine dieser Schichten ist der Virtual Functional Bus (VFB), der eine Zwischenschicht zwischen der Hardware und der Software darstellt. Er kann verglichen werden mit einer Middleware. Für jede AUTOSAR Komponente wird eine Instanz des Virtual Functional Bus generiert. Dies ist die Runtime Environment (RTE). Durch diesen Aufbau können Applikationen auf verschiedenen Steuergeräten untergebracht werden, beispielsweise wenn Funktionen von Fahrerassistenzsystemen auf unterschiedliche Steuergeräte verteilt sind. Die notwendige Kommunikation wird von der RTE automatisch geregelt. Sind beispielsweise zwei Applikationen auf einem Steuergerät untergebracht, kann die Kommunikation über gemeinsame Variablen erfolgen. Sind die Applikationen auf verschiedenen Steuergeräten untergebracht, so wird automatisch eine Kommunikation mittels eines Busses umgesetzt.

Neben der Einführung von standardisierten Schnittstellen beinhaltet der AUTOSAR Standard auch eine neue Entwicklungsmethodik. Diese sieht aber keine Analysephase vor. Vielmehr besagt sie, dass zunächst die einzelnen Software- (SWCs) und Hardwarekomponenten (ECUs) und anschließend das System an sich konfiguriert werden. Dies geschieht in verschiedenen Beschreibungsdokumenten sogenannte Descriptions (vgl. Abbildung 2). Somit entsteht die eben beschriebene Lücke zwischen Anforderungen und AUTOSAR Modell. Nach der Konfiguration kann durch verschiedene Generatoren der entsprechende Code erzeugt werden (siehe Abbildung 2). Wie erwähnt, gibt es aber keine Unterstützung für die werkzeuggestützte Analyse der notwendigen Konfigurationsdaten. Beispielsweise erfordert das Betriebssystem eine Konfiguration der einzelnen Tasks mit Laufzeiten und Prioritäten. Diese können einen großen Einfluss auf die Performance haben. Nachdem die Tasks mit diesen Informationen erzeugt wurden, können die einzelnen Funktionen von AUTOSAR (Runnables) auf die verschiedenen Tasks aufgeteilt werden. Fehler bei diesen Einstellungen können Performanceprobleme und sogar das Verletzen von Zeitanforderungen bedeuten.

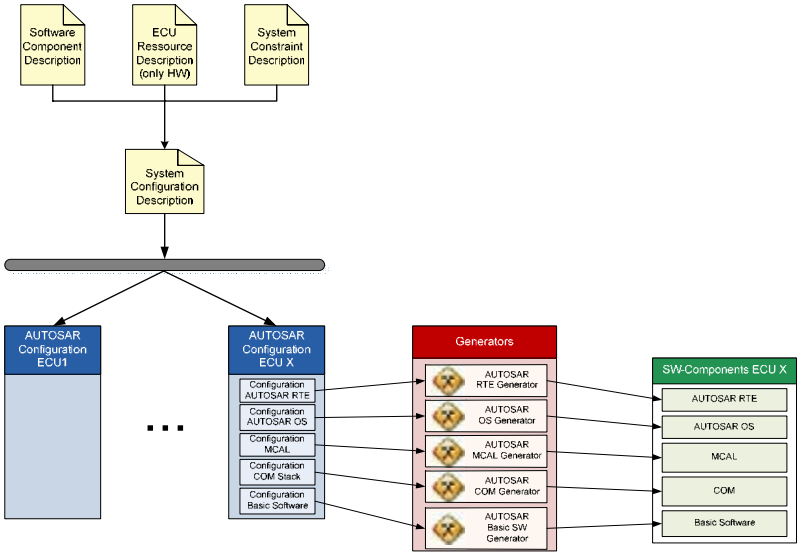


Abbildung 2: AUTOSAR Entwicklungsmethodik

Ebenso müssen auch die Basis-Softwarekomponenten und die Kommunikation (z.B. CAN- oder FlexRay-Bus) ausgewählt und konfiguriert werden. Hierbei ist darauf zu achten, dass die Kommunikation mit den anderen Teilen des Systems, besonders dem Betriebssystem, abgestimmt ist. Die notwendige Synchronisation kann beispielhaft am Senden von Daten erläutert werden. Das Senden erfolgt dabei häufig zyklisch, d.h. zu bestimmten Zeitpunkten werden die Daten über den Bus versendet. Von daher muss der Kommunikationsbus zu dem Zeitpunkt bereit sein, wenn die sendende Funktion vom Betriebssystem aktiviert wird. Ist eine Abstimmung nicht gegeben, kann es zu Verzögerungen bei der Kommunikation kommen, z.B. weil der Bus bereits belegt ist oder noch nicht bereit ist und es somit zu Wartezeiten kommen kann. Dies kann soweit gehen, dass zeitliche Anforderungen, beispielsweise Fristen bei der Kommunikationsübertragung, nicht eingehalten werden.

3 Das Analysemodell

In dem hier vorgestellten Ansatz werden die typischerweise, wie oben erwähnt, nur informal vorliegenden Anforderungen zunächst in Form von Use-Cases dargestellt, die dann durch Aktivitäts- und Sequenzdiagramme weiter verfeinert werden. Hierdurch ist es möglich, einzelne Szenarien darzustellen, die das zu entwickelnde System erfüllen soll. Aus den in dieser Analyse gewonnenen Daten, lässt sich eine Architektur in Form von Komponenten- und Klassendiagrammen erstellen. Das Verhalten der einzelnen Komponenten der Architektur wird dann durch eine spezielle Form von zeitbehafteten Zustandsdiagrammen, die aus den einzelnen Szenarien abgeleitet werden, modelliert. Diese erlauben es, durch automatische Überprüfung (model checking) zeitliche Inkonsistenzen festzustellen und bilden die Basis für das oben erwähnte Scheduling der einzelnen Tasks sowie die anschließende Konfiguration der Architektur und die automatische Codegenerierung.

Gerade die korrekte Umsetzung der zeitlichen Restriktionen (Deadlines) der einzelnen Funktionen aus der Anforderungsdefinition ist eine wesentliche Voraussetzung dafür, dass aus dem AUTOSAR-kompatiblen Modell eines Systems korrekter Code z.B. für die RTE generiert werden kann. Die Einhaltung der zeitlichen Restriktionen drückt sich insbesondere in der Konfiguration der Tasks aus. Eine Reihe darüber hinausgehender Parametrisierungsmöglichkeiten in AUTOSAR können wir hier aus Platzgründen nicht weiter betrachten.

Zeitbehaftete Modellierungsansätze, insbesondere Automaten, die die Modellierung von Deadlines, Mindestzeitdauern oder Worst-Case Execution Times (WCET) unterstützen, sind durch sogenannte Profile wie SPT (Schedulability, Performance and Time) und MARTE in der UML vorhanden. Diese Profile bieten aber keine Möglichkeit, die entstandenen Modelle mangels einer präzisen semantischen Definition automatisch auf Korrektheit und Konsistenz zu überprüfen, sowie „Scheduling“ und Codegenerierung anzuschließen.

Deshalb basiert unser Ansatz auf einer speziellen Erweiterung der „Timed Automata“, den sogenannten Realtime Statecharts (RT-Statecharts) [GTB+03], die es erlauben, gerade zeitbehaftete Modelle in einer sehr kompakten Form darzustellen. Darüber hinaus lässt sich dann ein existierender leistungsfähiger „Model Checker“ (UUPAAL) zur Analyse einsetzen [HH07], sowie ein existierendes Scheduling-Verfahren entsprechend an AUTOSAR anpassen [BGS05]. Letztlich ist dieser Ansatz in eine umfassende Entwicklungsumgebung zur Modellierung, Analyse und Implementierung mechatronischer Systeme, der MechatronicUML, eingebettet [BGH+07] [GBSO04].

Beispielhaft für die Modellierung einer Systemfunktionalität mit Realtime Statecharts wird in Abbildung 3 die (stark vereinfachte) Kamerasteuerung eines Fahrerassistenzsystems beschrieben. Nach der Initialisierung erfolgt eine Objektsuche in den aufgenommenen Bildern. Diese muss nach Verlassen des Initialisierungszustands innerhalb von 5 ms erreicht werden. Dies wird durch Angabe der WCET am Übergang modelliert. Falls ein vorher gespeichertes Objekt erkannt wird, dann wird dieses in einem Display zur Kenntnisnahme des Fahrers dargestellt.

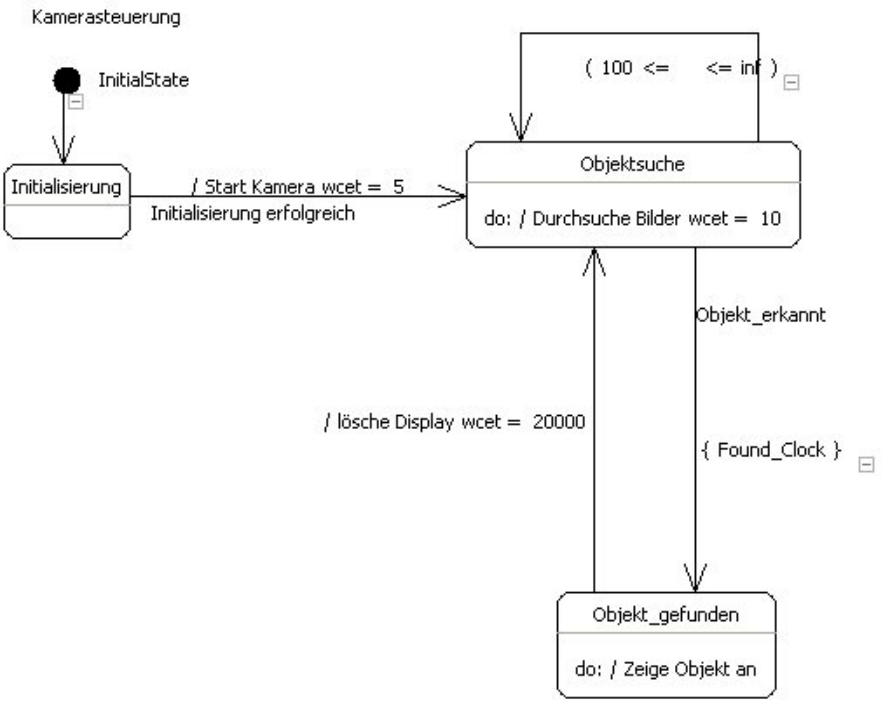


Abbildung 3: Kamerasteuerung RT-Statechart

Auf Basis der für alle Komponenten vorliegenden zeitbehafteten Verhaltensmodelle lässt sich eine Scheduling Analyse über alle Komponenten durchführen. Die Analyse berechnet automatisch die Reihenfolge der einzelnen Funktionen [BGT05], wobei die Abhängigkeiten untereinander berücksichtigt werden. Infolgedessen können den einzelnen Funktionen Prioritäten zugeordnet werden, die der Reihenfolge ihrer Abarbeitung entsprechen. In den Steuergeräten und den darauf befindlichen Betriebssystemen werden meist zeitscheibenbasierte Konfigurationen verwendet. Innerhalb der Zeitscheiben können dann Tasks ausgeführt werden. Die Reihenfolge der Tasks ist dabei abhängig von der Priorität. Es bietet sich somit an, die in der Analyse berechneten Prioritäten zu übernehmen.

Auch bei der Verteilung von Applikationsfunktionen auf verschiedene Steuergeräte müssen die im Analysemodell festgelegten zeitlichen Anforderungen eingehalten und dementsprechend bei der Definition der verschiedenen Tasks und deren Priorisierung berücksichtigt werden. Zurzeit wird die Verteilung mittels Erfahrungswissen und implizitem Wissen über das Modell von den Systemintegratoren manuell erstellt [KeS07]. Dies Verfahren hat aber den Nachteil, dass wegen der Vielzahl von Abhängigkeiten und der damit einhergehenden Komplexität eine vollständige Übersicht nicht mehr gegeben ist. Demzufolge ist es fast unmöglich, manuell eine optimale Reihenfolge und eine Verteilung der Funktionen auf die Tasks zu finden. Eine suboptimale Reihenfolge kann zu zeitlichen Abweichungen und schlimmstenfalls zu Systemfehlern führen.

Daher ist es sinnvoll, die automatisch generierten Reihenfolgen und Prioritäten aus dem Analysemodell zur Konfiguration der AUTOSAR-kompatiblen Architektur wiederzuverwenden. Hierzu muss eine Abbildungsvorschrift erstellt werden, in der die einzelnen Funktionen des Analysemodells auf AUTOSAR-Softwarekomponenten abgebildet werden. Diese Softwarekomponenten beinhalten eine oder mehrere sogenannte Runnables, welche ausführbare Einheiten in AUTOSAR sind. Eine Funktion wird typischerweise auf ein Runnable abgebildet, die durch ein RTE-Event ausgelöst wird [Auto08]. Dieser Abbildungsschritt muss weiterhin manuell erfolgen, während aber alle folgenden Schritte jetzt automatisiert aus dem Analysemodell abgeleitet werden.

Die einzelnen Runnables werden auf die Tasks des Betriebssystems aufgeteilt. Hierbei ist eine 1:n Verteilung möglich, d.h. mehrere Runnables können einer Task zugeordnet werden. Die Schwierigkeit besteht darin die Runnables so auf die Tasks zu verteilen, dass mit einer minimalen Anzahl von Tasks und deren zyklischen Aktivierung die zeitlichen Abhängigkeiten und somit auch die zeitlichen Anforderungen eingehalten werden. Hierzu werden die Ergebnisse der Scheduling Analyse aus den RT-Statecharts (Priorität und zeitliche Reihenfolge) weiterverwendet, da sie die Abhängigkeiten und Prioritäten bereits beinhalten. Zurzeit wird von einer zyklischen Aktivierung der Tasks ausgegangen. Es wird aber ferner untersucht, ob weitere Aktivierungsformen mit dem Ansatz modelliert werden können.

Der AUTOSAR Standard bedarf im Bereich des Scheduling [SRT+05] und der zeitlichen Informationen [ROH+08] noch Verbesserungen. Das hier skizzierte Analysemodell kann als Ansatz verstanden werden, diesen Schwächen zu begegnen und vor allem ein dafür notwendiges aber nicht vorhandenes einheitliches Zeitmodell zur Verfügung zu stellen.

4 Verwandte Arbeiten

Es existieren zahlreiche Arbeiten, die im Kontext des Standards AUTOSAR entstanden sind. Diese beziehen sich weitestgehend auf die Eigenschaften des Standards oder aber auf die Modellierung der Zwischenschicht, der sogenannten Runtime Environment, vgl. z.B. [POF+05]. In [ROH+08] werden die Zeitprobleme von AUTOSAR herausgestellt. Im gleichen Problemfeld arbeitet das Timmo Projekt und versucht ein einheitliches Zeitmodell für AUTOSAR zu entwickeln [Ri08] [Jer07]. Bei all diesen Arbeiten wird aber nicht auf die Frage eingegangen, wie die Informationen für die Architektur und die Konfiguration formal präzisiert und analysiert werden können und wie ein darauf basierender Ansatz für die automatische Codegenerierung aussieht. Die Arbeit [NWW+06] beschäftigt sich mit der Verteilung von AUTOSAR Funktionen, aber die Abhängigkeiten müssen explizit in Prolog modelliert werden, bevor sie mittels eines Constraint Solvers gelöst werden können. Diese Methode löst das Problem der Verteilung der verschiedenen Softwarekomponenten auf die verschiedenen Steuergeräte. Hier werden jedoch nicht die Konfiguration des Betriebssystems und die zeitlichen Abhängigkeiten der einzelnen Funktionen untereinander betrachtet.

Desweiteren beschäftigen sich Arbeiten mit dem allgemeinen Problem der Verteilung und Konfiguration. So werden in [HoN05] und [OMG03] die Sprachen eODL und DnC miteinander verglichen, die eine Verteilung und Konfiguration von verteilten komponentenbasierten Elementen ermöglichen. Diese beiden Modellierungssprachen können jedoch nicht für den AUTOSAR Standard übernommen werden. Bei der Sprache eODL verhindert die Möglichkeit zur Modellierung von Hierarchien den Einsatz. Während bei der Sprache DnC die Voraussetzung einer abgeschlossenen Spezifikation die Verwendung in der Designphase verhindert.

5 Zusammenfassung und Ausblick

Die manuelle Ermittlung von AUTOSAR-Konfigurationsdaten ist mühsam und auch fehlerträchtig. Vor allem ist es bei der Konfiguration von AUTOSAR schwierig eine ressourcenoptimale Verteilung der Software auf die verschiedenen Betriebssystemtasks vorzunehmen. Darum wird in diesem Positionspapier eine (halb)-automatische Konfiguration vorgeschlagen, die bereits in den frühen Phasen des Entwicklungsprozesses für valide Modelle sorgt. Die Konfiguration betrachtet im Moment im Wesentlichen die Verteilung der Funktionen auf Tasks. Jedoch wird an einer Ausweitung auf andere notwendige Konfigurationen gearbeitet.

Für die werkzeugunterstützte Konfiguration wurde zunächst dargelegt, dass ein Analysemodell erstellt werden muss, um die Lücke zwischen den informal formulierten Anforderungen und dem AUTOSAR Modell zu schließen. In dem Analysemodell werden die Anforderungen präzise modelliert und auf Korrektheit und Konsistenz analysiert. Im Vordergrund stehen dabei nichtfunktionale Anforderungen, die insbesondere die Festlegung von zeitlichen Restriktionen betreffen. Diese werden durch Hardware spezifische Informationen ergänzt, die für die Berechnung der zeitlichen Abhängigkeiten (z.B. WCET) benötigt werden. Die Informationen werden in dem Analysemodell mittels einer Erweiterung der Modellierungssprache UML (MechatronicUML) modelliert und analysiert. Aus der Analyse lassen sich z.B. Taskabhängigkeiten und Prioritäten ermitteln. Diese dienen als Konfigurationsdaten für das AUTOSAR Modell. Der Transfer zu AUTOSAR erfolgt dabei durch Abbildungsvorschriften.

Der hier vorgestellte Lösungsansatz ist noch zu vervollständigen. Es wurden bisher nur Konfigurationsdaten für die Verteilung der Funktionen auf Betriebssystemtasks werkzeugunterstützt gewonnen. Es ist zu untersuchen, ob auch noch weitere Konfigurationsdaten aus dem Analysemodell gewonnen werden können. Ein Ansatzpunkt ist z.B. die Kommunikation. Für den AUTOSAR Standard müssen die einzelnen Signale den verschiedenen Funktionen zugeordnet werden. Hier ist ein ähnliches Problem, wie bei der Verteilung der Funktionen, zu finden. Daher ist zu vermuten, dass auch hierfür die Informationen aus dem Analysemodell verwendet werden können.

Aber auch die werkzeugunterstützte Gewinnung von Konfigurationsdaten für die Verteilung der Funktionen kann noch verbessert werden. So ist zu untersuchen, welche Auswirkungen die verschiedenen Kommunikationsformen von AUTOSAR (beispielsweise explizite und implizite) auf das zeitliche Verhalten haben. Diese müssen bei der Übertragung der Ergebnisse vom Analysemodell nach AUTOSAR berücksichtigt werden. Somit können die im Analysemodell gewonnenen Daten weiter genutzt werden und der Entwickler wird bei seiner Arbeit deutlich entlastet.

Literaturverzeichnis

[Auto08] Der AUTOSAR-Standard, http://www.autosar.org, November 2008.

[BGS05] Burmester, S.; Giese, H.; Schäfer, W.: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code, in Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany, vol. 3748 of Lecture Notes in Computer Science (LNCS), pp. 25--40, Springer Verlag, November 2005

[BGT05] Burmester, S.; Giese, H.; Tichy, M.: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML, Model Driven Architecture: Foundations and Applications, Springer-Verlag, 2005.

[Bro06] Broy, M.: Challenges in Automotive Software Engineering, International Conference on Software Engineering (ICSE), 2006.

[BGH+07] Burmester, S.; Giese, H.; Henkler, S.; Hirsch, M.; Tichy, M.; Gambuzza, A.; Much, E.; Vöcking, H.: Tool Support for Developing Advanced Mechatronic Systems: Integrating the Fujaba Real-Time Tool Suite with CAMeL-View, in Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA, pp. 801--804, IEEE Computer Society Press, May 2007.

[GBSO04] Giese, H.; Burmester, S.; Schäfer, W.; Oberschelp, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration, Proceedings12th ACM SIGSOFT Foundations of Software Engineering (FSE), 2004

[GTB+03] Giese, H.; Tichy, M.; Burmester, S.; Schäfer, W.; Flake, S.: Towards the Compositional Verification of Real-Time UML Design, Proceedings of the European Software Engineering Conference (ESEC), 2003.

[HKK04] Hardung, B.; Kölzow, T.; Krüger, A.: Reuse of Software in Distributed Embedded Automotive Systems, EMSOFT 2004.

[HH07] Henkler, S.; Hirsch, M.: Compositional Validation of Distributed Real Time Systems, in Proc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany, 30.-31.10.2007 (Matthias Gehrke, Holger Giese, and Joachim Stroop, eds.), vol. tr-ri-07-286, pp. 52--56, University of Paderborn, October 2007.

[HoN05] Hoffmann, A.; Neubauer, B.:Deployment and Configuration of Distributed Systems, SAM 2004, LNCS 3319 Springer Verlag, 2005.

[Jer07] Jersak et.al.: Timing model and methodology for AUTOSAR. Elektronik Automotive. October 2007

[KeS07] Kebemou, A.; Schieferdecker, I.: Evaluating Modeling Solutions on Their Ability to Support the Partitioning of Automotive Embedded Systems, Emerging Directions in Embedded and Ubiquitious Computing, Springer Verlag, 2007.

[NWW+06] Nechypurenko, A.; Wuchner, E.; White, J.; Schmidt, D. C.: Applying Model Intelligende Frameworks for Deployment Problem in Real-Time and Embedded Systems, Conference on Model Driven Engineering Languages and Systems Models, 2006.

[OMG03] Deployment and Configuration of Component-based Distributed Applications Specification, OMG Adopted Specification, http://www.omg.org/docs/ptc/03-07-08.pdf, Juli 2003.

[POF+05] Pelz, G.; Oehler, P.; Fourgeau, E.; Grimm, C.: Automotive System Design and AUTOSAR, Advances in Design and Specification Languages for SoCs, 2005.

[WTS08] Webers, W.; Thörn, C.; Sandkuhl, K.: Connecting Feature Models and AUTOSAR: An Approach Supporting Requirements Engineering in Automotive Industries, Proc. 14th International Working Conference on Requirements Engineering (REFSQ), 2008.

[Ri08] Richter, K.: Defining a Timing Model for AUTOSAR – Status and Challenges. Software Engineering Conference, Workshop Automotive Software Engineering: Forschung, Lehre, Industrielle Praxis. Munich, Germany. February 2008

[ROH+08] Rudorfer, M;, Ochs, T; Hoser, P.; Thiede, M; Mössmer, M; Scheickl, O.; Heinecke, H.: Echtzeitfähigkeit ist alles - Echtzeit-Systemdesign mit Hilfe der AUTOSAR-Methodik, WEKA – Fachmedien, 2008.

[SKF07] Siwy, R.; Kloss, N.; Fürst, S.; Schmid, H.: Modellothek: Modellbasierte Entwicklung von wieder verwendbaren AUTOSAR SW-Komponenten für Karosserie-Funktionen, VDI Berichte Nr. 1907, Baden-Baden, 2007.

[SRT+05] Salzmann, C.; Rudorfer, M.; Thiede, M.; Ochs, T.; Hoser, P.; Mössmer, M.; Heinecke, H.; Münnich, A.: Erfahrungen mit der technischen Anwendung einer AUTOSAR Runtime Environment, VDI Berichte Nr. 1907, Baden-Baden, 2005.

[Win05] Windpassinger, H.: AUTOSAR Methodology – Erfolgskonzept für die Kfz-Software Entwicklung, Automotive Journal, 2005.

[WTS08] Webers, W.; Thörn, C.; Sandkuhl, K.: Connecting Feature Models and AUTOSAR: An Approach Supporting Requirements Engineering in Automotive Industries, Proc. 14th International Working Conference on Requirements Engineering (REFSQ), 2008

Modellbasierte Entwicklung in der Prozessautomatisierung

- Übersicht-

Ulrich Epple

Lehrstuhl für Prozessleittechnik
RWTH Aachen
Turmstr. 46
52064 Aachen
epple@plt.rwth-aachen.de

Abstract: Der Beitrag gibt eine Übersicht über die Modellierungsvorstellungen in der Prozessautomatisierung. Am Beispiel der Basis-Automatisierungsfunktionen werden die klassischen Strukturierungs- und Entwurfsverfahren erläutert. Die strengen Anforderungen an Echtzeit-Zuverlässlichkeit, Robustheit und einfache Handhabung im laufenden Betrieb sind nur auf der Grundlage eines äußerst restriktiven modellbasierten Entwurfsprozesses möglich. Der Beitrag gibt eine Übersicht über die grundlegenden Konzepte. Im zweiten Teil werden neue Ansätze zur Erweiterung der Modellbasis, zu ihrer Formalisierung und zur Entwicklung eines regelbasierten und partiell automatisierbaren Entwurfsprozesses diskutiert. Die Modelle und Entwurfsvorgänge werden an Demos erläutert

1 Systemstrukturen in der Prozessautomation

In der Prozesstechnik sind die leittechnischen IT-Systeme traditionell nach einem Ebenenmodell gegliedert. Auf der untersten Ebene, der Feldebene, befinden sich die Messgeräte (Sensoren) und die Aktor-Steuergeräte. Diese als Feldgeräte bezeichneten Komponenten besitzen heute typischerweise einen leistungsstarken µ-Controller und die Fähigkeit digital zu kommunizieren. Sie sind über ein Feldbussystem untereinander und mit den Automatisierungskomponenten verbunden. In der Prozessindustrie sind heute insbesondere die Feldbussysteme ProfibusPA, FoundationFieldbusFF und HART verbreitet. Oberhalb der Feldebene kommt die Prozessleitebene. Sie besteht aus den dezentralen Komponenten, den zentralen Komponenten und dem Systembus. Die dezentralen Komponenten sind die eigentlichen Automatisierungskomponenten. In ihnen werden die Basis-Automatisierungsfunktionen realisiert. An die Automatisierungs-komponenten werden hohe Anforderungen bezüglich Echtzeitverhalten, Unterstützung von Redundanzstrukturen, Rekonfigurierbarkeit im laufenden Betrieb, einfache Handhabbarkeit, Langzeitverfügbarkeit und Robustheit gegenüber Umgebungseinflüssen gestellt.

Um dies zu gewährleisten, bauen die Automatisierungskomponenten renommierter Hersteller heute immer noch auf spezieller Hardware und speziellen Betriebssystemfunktionen auf. Die Aufgabe einer Automatisierungskomponente ist die autarke Führung und Überwachung einer Teilanlage oder eines Anlagensegments. In einer Produktionsanlage benötigt man schon aus Dezentralitätsgründen mehrere Automatisierungskomponenten. Diese kommunizieren auf der einen Seite über den Feldbus mit den Feldgeräten und auf der anderen Seite über den sogenannten Systembus mit anderen Automatisierungskomponenten und zentralen Komponenten. Die zentralen Komponenten dienen der Bereitstellung von zentralen leittechnischen Funktionen, z.B. einem Verlaufsarchiv, den Operator-Bedienoberflächen, dem Melde- und Alarmsystem usw. In der Prozessleittechnik rechnet man auch die Engineering- und Programmiersysteme zu den zentralen Komponenten. Sie sind also integraler Bestandteil des Prozessleitsystems. Oberhalb der Prozessleitebene befinden sich die MES-Ebene (Manufacturing Execution System) und die ERP-Ebene (Enterprise Ressource Planning). Diesen Ebenen werden alle IT-Funktionen zur Führung des Betriebs, der Produktion und des Unternehmens zugeordnet. In diesem Beitrag sollen die Konzepte zum Entwurf der Anwenderfunktionalität auf der Feldebene und der Prozessleitebene betrachtet werden. Aus Sicht der Informatik sind Feldgeräte und Automatisierungskomponenten über ein Kommunikationssystem vernetzte eingebettete Systeme.

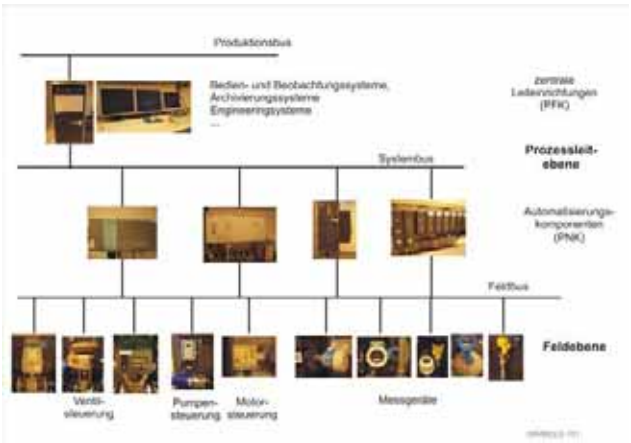


Bild 01: Aufbau der Feld- und der Prozessleitebene

2 klassische Modellansätze als Entwurfsgrundlage

In der Prozessautomatisierung ist jede Anlage ein Unikat. Die Loszahl auf Anlagenebene ist praktisch gleich eins. Dies erfordert ein hocheffizientes Engineering. Auf der anderen Seite wird in der Prozessautomatisierung ein hohes Maß an Zuverlässigkeit und Fehlerfreiheit erwartet. Insgesamt sind diese Forderungen nur auf der Grundlage eines formalisierten modellbasierten Entwurfs zu erreichen. In der Prozessautomatisierung hat sich als Entwurfsgrundlage das *Funktionsbausteinmodell* weltweit etabliert.

Dieses Modell bildet die Aufbaustruktur der klassischen, auf Einzelgerätetechnik basierenden Zentralwarten funktional ab. In der Folge hat sich das Funktionsbausteinmodell zu einem von der Realisierung unabhängigen Entwurfsansatz herausgebildet und ist heute das dominante Modellierungskonzept in der Prozessautomatisierung. Die Unterstützung dieses Konzepts durch die Entwurfs- und Betriebssystemplattformen der rechnergestützten Prozessleitsysteme war eine der wesentlichen Voraussetzungen für die erfolgreiche Einführung dieser neuen Technologie.

2.1 historische Wurzeln

Die wesentlichen Grundzüge des Funktionsbausteinmodells lassen sich anschaulich am Beispiel des Einzelgerätemodells erläutern. In Bild 2 ist eine Anlagensteuertafel dargestellt, wie sie in den Zentralwarten zu Zeiten der Einzelgerätetechnik (ca 1970) weltweit verbreitet war. Die Abbildung zeigt die Vor- und die Rückseite der Tafel. In der funktionalen Analogie entspricht jedes Gerät einem Funktionsbaustein.

Bild 2: Die Modellvorstellungen der Einzelgerätetechnik

Aus der Darstellung ergeben sich direkt die wesentlichen Eigenschaften des Funktionsbausteinmodells.

- Die Gesamtfunktionalität ergibt sich aus dem Zusammenwirken der Einzelgeräte.
- Jedes Einzelgerät arbeitet für sich, vollständig unabhängig von allen anderen Geräten.

- Jedes Einzelgerät führt seine Funktion quasikontinuierlich aus.
  Jedes Einzelgerät berechnet aus den aktuell anliegenden Eingangswerten und seinen Zustandswerten ständig neue Zustandswerte und stellt diese als Ausgangswerte zur Verfügung. Die "ständige" Berechnung kann sowohl durch analoge Schaltkreise als auch durch ein z.B. in einem schnellen Zyklus auf einem µ-Prozessor laufendes Programm realisiert werden. Mit der Bezeichnung "quasikontinuierlich" wird ausgedrückt, dass die durch die diskrete Realisierung entstehenden zeitdiskreten Aspekte anwendungsseitig nicht berücksichtigt werden müssen und dass das äußere Verhalten des Geräts funktional als kontinuierlich angesehen werden kann.

- Die Kommunikation zwischen den Geräten erfolgt durch Signalverbindungen.

- Signalverbindungen werden einzeln und unabhängig von den Geräten hantiert.

- Signalverbindungen realisieren ein rückwirkungsfreies Lesen.

2.2 Das Funktionsbausteinmodell

Abstrahiert man die Einzelgeräte auf ihre automatisierungstechnische Funktionalität, dann erhält man das Funktionsbausteinmodell. Der Grundaufbau des der Schalttafel entsprechenden Funktionsbausteinschemas ist in Bild 3 dargestellt.

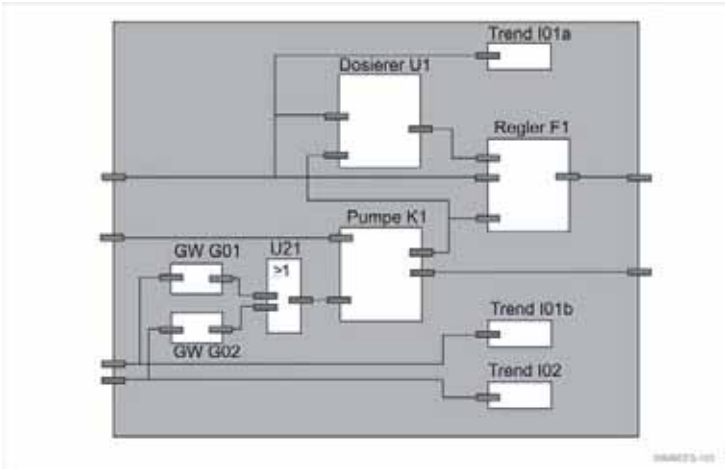


Bild 3: Das Funktionsbausteinmodell

Funktionsbausteine besitzen Eingänge, interne Bausteinzustände, Ausgänge und eine quasikontinuierlich arbeitende Methode. Die Bausteinmethode berechnet ständig aus den an den Eingängen anliegenden Werten und den Werten der internen Bausteinzustände neue Werte der internen Bausteinzustände und der Ausgänge. Jeder Baustein tut dies ständig, unabhängig vom Daten- und Kontrollfluss der Anwendung, unabhängig von der Existenz oder dem Bearbeitungszustand aller anderen Bausteine und unabhängig davon, ob die Eingänge verbunden sind oder nicht. Alle Bausteine arbeiten also grundsätzlich unabhängig und nebenläufig. Die Verbindungen sind eigene aktive Einheiten, die ständig dafür sorgen, dass der entsprechende Eingangswert gleich dem verbundenen Ausgangswert ist. Die Übertragung durch die Verbindungen erfolgt entsprechend dem Signalmodell rückwirkungsfrei und quasikontinuierlich.

2.3 Softwaretechnische Realisierung

In der Prozessautomatisierung ist es erforderlich, dass die Automatisierungsstruktur im laufenden Betrieb umkonfiguriert werden kann. Für eine nach dem Funktionsbausteinmodell entworfene Anwendung bedeutet dies, dass im laufenden Betrieb sowohl Verbindungen als auch Funktionsbausteine jederzeit gelöscht oder neu hinzugefügt werden können, ohne die Funktionsausführung der nicht betroffenen Bausteine und Verbindungen zu stören. Bausteine müssen also als Softwaremodule realisiert sein, die sich als Ganzes nebenwirkungsfrei hantieren lassen, die im eingebauten Zustand persistente Zustände besitzen und zu jedem Zeitpunkt an ihren Ausgängen gültige Werte bereitstellen. Jeder Baustein beansprucht einen definierten Speicherplatz und eine definierte Rechenleistung. Durch die bereitgestellten Entwurfsmittel wird sichergestellt, dass der definierte Ressourcenbedarf in keinem Fall überschritten wird. Es ist Aufgabe des leittechnischen Betriebssystems den Bausteinen die vereinbarten Ressourcen (zyklische Zeit-Slots, Speicherbereiche) zuzuteilen und deren Einhaltung zu sichern.

2.4 Das Typ-Instanzkonzept

Die Implementierung eines Funktionsbausteins erfolgt in einem zweistufigen Prozess: Der Implementierung der Bausteinklasse als Typobjekt und der Implementierung einer Instanz dieses Typs als Instanzobjekt. Im Gegensatz zur Einzelgrätetechnik genügt es in der softwaretechnischen Realisierung ein bestimmtes Typobjekt in einer Prozessumgebung nur einmal zu implementieren. Diesem Typobjekt können dann beliebig viele Instanzobjekte zugeordnet werden. Das Typobjekt verwaltet die Bausteinmethode, die Struktur der Instanzen (Eingänge, Ausgänge, Zustände) und den Ressourcenbedarf. Für die Spezifikation des Typobjekts sieht das klassische Funktionsbausteinmodell vier unterschiedliche Möglichkeiten vor:

1. black box
Der Bausteintyp wird als vordefiniertes Bibliothekselement (i.A. vom Hersteller) zur Verfügung gestellt. Die Art seiner Realisierung bleibt verborgen. Der Hersteller garantiert die korrekte Funktion und die Einhaltung des Ressourcenbedarfs. Es gibt eine Reihe von Leitsystemen, die nur die Verwendung von vordefinierten Bibliotheks-elementen zulassen.

2. Strukturierter Text (ST), Anweisungsliste (AWL)
Strukturierter Text und Anweisungsliste sind einfache in der IEC 61131-3 standardisierte textuelle Sprachen zur Beschreibung der Struktur und der Methode von Funktionsbausteinen. Sie erlauben eine "freie" Programmierung in eingeschränktem Rahmen (keine Pointer, keine Schleifen..).

3. ContinuousFunctionChart (CFC), Kontaktplan(KOP,LD)
Das Funktionsbausteinmodell lässt es zu, dass die Beschreibung eines Netzwerks aus bereits definierten einfacheren Bausteinen zur Spezifikation eines neuen komplexen Bausteintyps verwendet wird. (IEC61131-3). Für einfache binäre Logiken (z.B. die häufig benötigten Verriegelungslogiken) werden die Bausteintypen durch spezielle Graphik-Icons gekennzeichnet. Dabei ist sowohl eine symbolische Blockgraphik als auch eine als Kontaktplan bezeichnete Graphik, die sich an der Darstellung der elektrischen Schaltelementen orientiert, standardisiert.

4. SequentialFunctionChart (SFC)
Das Funktionsbausteinmodell lässt es zu, dass die Methode eines Funktionsbausteins in Form eines speziellen Zustandsautomaten (Ablaufkette) beschrieben wird.

2.5 Keine Unterstützung abstrakter Klassen im operativen System

In der Prozessautomatisierung werden Abstraktionsprinzipen angewendet, um z.B. in CAE-Systemen Auswahlhierarchien zu gliedern oder im Planungsverlauf Festlegungen schrittweise genauer zu fassen, sie sind jedoch nicht formalisiert und bilden sich nicht in die operativen Strukturen ab. Benötigt man z.B. für eine Temperaturregelung einen Reglerbaustein, dann wird man im Verlauf der Planung die Reglerart zwar schrittweise aus einer allgemeinen Anforderung (Regler) festlegen, im Prozessleitsystem gibt es als Bausteintypen jedoch nur die konkreten Reglerklassen. Das Leitsystem weiß also nicht, dass die Bausteintypen "PID" und "Zweipunkt" beides Regler sind. Das Funktionsbausteinmodell unterstützt die Bildung von abstrakten Klassen nicht.

3 Neue Modellansätze als Grundlage eines optimierten Entwurfs

Die Prozessautomatisierung befindet sich in einem starken Wandel. Neue, über die Basisautomatisierung hinausgehende Funktionsanforderungen werden in erheblichem Maß an sie herangetragen und führen zu einer signifikanten Erweiterung der zu implementierenden Funktionalität. Asset-Management, Performence Monitoring, Produktverfolgung, Supply-Chain-Unterstützung, Prozessoptimierung, Life-Cycle-Dokumentation usw. können als Stichworte genannt werden. Diesen Anforderungen muss durch eine Optimierung des Entwurfsprozesses begegnet werden. Ziel dieser Optimierung ist sowohl eine signifikante Senkung des Engineeringaufwands als auch die weitere Qualitätsverbesserung der erstellten Software. Eine einfache Weiterentwicklung der bestehenden Modelle und Werkzeuge reicht nicht aus, um die gesteckten Ziele zu erreichen. Die erwarteten Verbesserungen sind zu groß und erfordern einen grundsätzlich neuen Entwurfsansatz. Ein Lösungsweg ist die Formalisierung und Automatisierung des Entwurfsprozesses selbst. Unter dem Stichwort *Automatisierung der Automatisierung* werden zur Zeit Konzepte diskutiert, wie ein solcher Ansatz für die Prozessautomatisierung aussehen könnte. In den folgenden Unterkapiteln sollen einige Vorschläge zur Gestaltung einer Modelllandschaft für die Prozessautomatisierung vorgestellt werden.

3.1 Struktur der Modelllandschaft

Um die Vielzahl der Aspekte, die beim Entwurf eines Prozessautomatisierungssystems zu berücksichtigen sind zu erfassen, bietet es sich an, eine modellbasierte Entwicklung nicht auf einem Modell aufzubauen, sondern eine Modelllandschaft aus einander ergänzenden und aufeinander aufbauenden Modellen zu entwickeln. In Bild 4 ist ein Ausschnitt aus einer solchen Modelllandschaft dargestellt. Farbig hinterlegt sind die Modelle, die am Lehrstuhl für Prozessleittechnik zur Zeit explizit als Grundlage des Entwurfsprozesses eingesetzt werden.

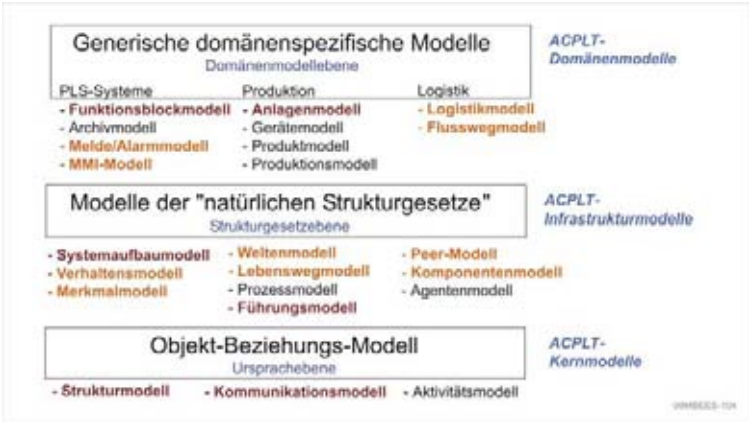


Bild 4: Ausschnitt aus der Modelllandschaft der Prozessautomatisierung

Beim Design einer solchen Modelllandschaft kann man sich von folgenden Aspekten leiten lassen: Die Modelle sollen technologieunabhängig sein und möglichst zeitlos zutreffende Zusammenhänge beschreiben (also "*wahr*" sein). Sie sollten redundanzfrei formuliert sein und entweder aufeinander aufbauen oder einander ihre Funktionalität als Dienste anbieten. Sie sollten im Anwendungsgebiet konsensfähig sein und sich für eine Standardisierung eignen. Sie sollten formal spezifiziert sein und als erkundbare Modellstrukturen im operativen System sich selbst erklären.

3.2 Metamodellbasierte Spezifikation

Die Beschreibung der Modelle sollte auf der Grundlage eines einfachen gemeinsamen Metamodells und der bereits spezifizierten Modelle erfolgen. Auf diese Weise ergibt sich ein Modellhierarchiebaum, aus dem heraus jederzeit neue Modelle entwickelt und hinzugefügt werden können.

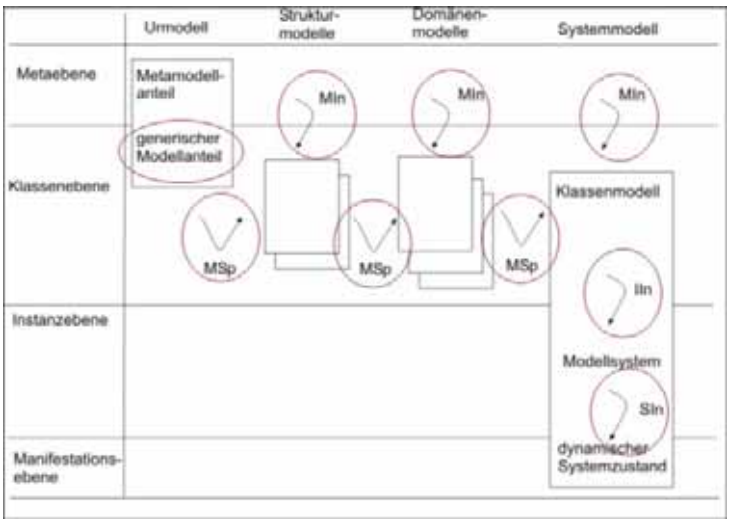


Bild 5: Modellebenen des am Lehrstuhl verfolgten ACPLT-Konzepts

In Bild 5 ist dargestellt, wie von links nach rechts die Modelle auf Klassenebene ausgeprägt werden. Ausgangspunkt ist die generische Klasse *Object* als Teil des Urmodells. Ausgehend von dieser Urklasse lassen sich abgeleitete Klassen spezifizieren (MSp) die selbst jeweils wieder Instanzen der Metaklasse *metaclass* sind (MIn). Im Klassenmodell der Anwendung stehen schließlich alle zur Lösung des Anwendungsproblems benötigten instanzierbaren Klassen bereit. Aus diesen lassen sich nun die zum Aufbau des konkreten Modellsystems benötigten Instanzen ableiten (IIn), verschalten und in den dynamischen Betrieb übernehmen (SIn). In Bild 6 ist das zugehörige Objektmodell dargestellt.

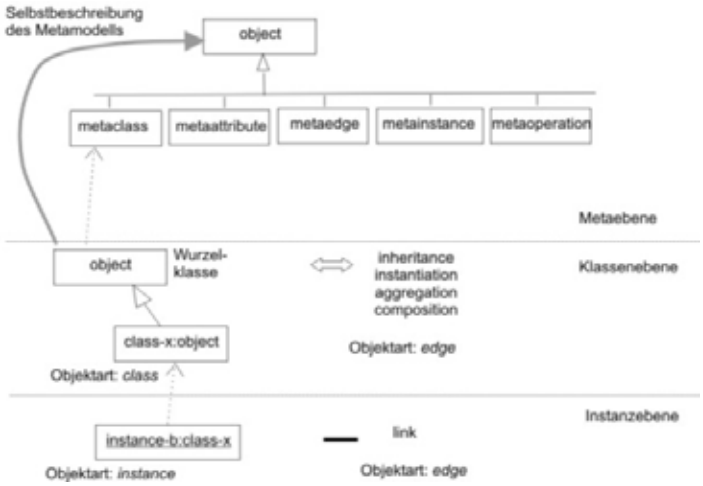


Bild 6: Modellstruktur des ACPLT-Konzepts.

3.3 Modellstruktur = Laufzeitstruktur

Ein entscheidendes Merkmal des hier vorgestellten Ansatzes ist die Abbildung der Modellstrukturen als explizite Objektstrukturen im operativen Systemnetzwerk. In sämtlichen Teilnehmern des Netzwerks (bis zum kleinen µ-Controller im Feldgerät) sind alle in Bild 5 und Bild 6 dargestellten Objekte und Beziehungen einschließlich der Metamodellebene explizit hinterlegt. Dies erlaubt es jeder Anwendung im Netzwerk jederzeit selbst lokal oder im Netzwerk Modelle zu erkunden und formal zu analysieren. Da die Funktionalität einer Instanz vollständig durch seine Klasse beschrieben ist, d.h. alle Methoden bekannt sind, besitzt das lokale Metamodell in allen Zielsystemen die Fähigkeit, Instanzen selbst lokal zu erzeugen. Instanzen werden also nicht geladen, sondern dynamisch durch das jeweilige Metamodell in den Zielsystemen auf Anforderung erzeugt. Diese aus der Historie der Funktionsbausteintechnik übernommene Vorstellung, Entwurfsmodelle 1:1 in den Laufzeitstrukturen abzubilden und operativ auf diesen Modellen zu arbeiten, ist ein entscheidender Schritt in Richtung auf eine automatisierte modellgetriebene Entwicklung. Schon auf dem derzeitigen Stand der Entwicklung können z.B. Instanznetzwerke durch Automaten im laufenden Betrieb lokal oder auf jeder beliebigen Netzwerkkomponente erstellt oder rekonfiguriert werden.

3.3 Ein Servicemodell für die leittechnischen Systemdienste

Es gehört zu den Vorzügen klassischer Prozessleitsysteme, dass diese ihre Anwendungsfunktionen durch leittechnische Betriebssystemfunktionen unterstützten. Leittechnische Betriebssystemfunktionen sind vom Hersteller in das System integrierte Funktionspakete, auf die eine Anwendung über Standardschnittstellen zugreifen kann. Typische Beispiele sind das Melde- und Alarmsystem, die Bedien- und Beobachtungsoberflächen, die Archivsysteme usw.. Die Schnittstellen und die Funktionalität der leittechnischen Systemfunktionen sind heute herstellerspezifisch ausgeprägt und auf die Verwendung in einer Ebene beschränkt.

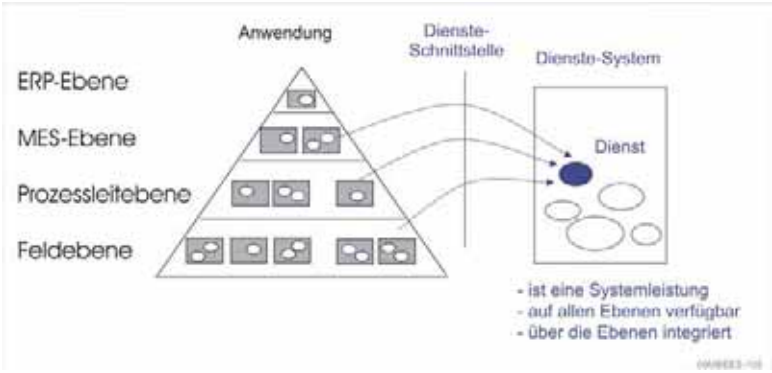


Bild 7: Servicemodelle für die leittechnischen Systemfunktionen

Aktuell sind Bestrebungen im Gange die leittechnischen Systemfunktionen wie in Bild 7 dargestellt als allgemeine Standarddienste zu spezifizieren und so den Anwendungen ebenen- und systemunabhängig zur Verfügung zu stellen. Dies würde auf Anwenderseite maßgeblich zu einer Vereinheitlichung der Entwurfsprozesse beitragen und den Herstellern die Möglichkeit bieten, sich durch robuste, leistungsfähige und autonome Systemmodule auszuzeichnen.

4. Zusammenfassung

Der Beitrag gibt einen Einblick in die Modellwelt und die Entwurfskonzepte der Prozessautomatisierung. Aus der historischen Entwicklung heraus und durch die speziellen Anforderungen hat sich in der Prozessautomatisierung ein eigenes Architekturmodell mit einer eigenen "Denke" entwickelt. In einem ersten Teil wird als Kernpunkt dieser Denke das klassische Konzept der Funktionsbausteintechnik erläutert und mit seinen Vor- und Nachteilen diskutiert. Das Funktionsbausteinmodell ist heute die Grundlage für die weltweit hohe Leistungsfähigkeit und Qualität des Softwareengineerings in der Prozessautomatisierung. In Zukunft genügt dies jedoch nicht. Durch den immens steigenden Informations- und Funktionsbedarf muss die Effektivität des Entwicklungsprozesses massiv gesteigert werden. Darüber hinaus erfordert die zunehmende Verzahnung von Zusatzfunktionen (wie Plant Asset Management, Performance Optimization..) über die Ebenen hinweg eine deutlich verbesserte vertikale Integration sowohl der Modelle als auch der Entwurfsprozesse, von den Feldgeräten über die Automatisierungskomponenten und die zentralen leittechnischen Komponenten hinweg bis zu den Produktions- und Betriebsleitsystemen auf der MES-Ebene. Benötigt wird also eine durchgängige Modellwelt und eine daraus abgeleitete durchgängige Entwurfs- und Laufzeitinfrastruktur, die sowohl den Anforderungen der eingebetteten Systeme als auch den zentralen Komponenten gerecht werden. Im Beitrag werden als Beispiele ein metamodellbasiertes Infrastrukturkonzept, ein erweitertes Funktionsbausteinmodell, erste Überlegungen zu einem serviceorientierten leittechnischen Betriebssystem und ein Ansatz zur Automatisierung der Automatisierung vorgestellt. Anhand von implementierten Lösungen werden die Konzepte demonstriert. Erfahrungen aus der industriellen Praxis zeigen das Potenzial der modellgestützten Entwicklung und Implementierung.

Partial Order Algorithms for Model-based Diagnosis of Discrete Event Systems

Dennis Klar Michaela Huhn

Technische Universität Braunschweig,
Institute for Software Systems Engineering,
Braunschweig, Germany
{d.klar, m.huhn}@tu-braunschweig.de

Abstract: Model-based diagnosis references explicit models of a system's structure and behaviour to explain observations. The reconstruction approach by Lamperti, Zanella et al. is especially well-suited to diagnose distributed systems of asynchronously communicating automata. It searches the global state space for traces consistent with sequences of messages output by the system. But even with sophisticated modularisation of the search process, the core algorithm heavily suffers from the interleaving state explosion problem. We present an adaption of model checking techniques based on partial orders and unfoldings together with a new reconstruction algorithm to overcome this problem. A scalable case study shows promising results.

1 Introduction

On-line diagnosis of large and complex technical systems is an important aspect of maintenance. Early detection of malfunctions and computer-aided localisation and identification of underlying faults help to reduce the time to restoration and therefore increase the system's overall availability. Model-based diagnosis uses explicit knowledge of a system's structure and behaviour in a white-box approach. This allows for direct conclusion from observed abnormal behaviour to affected components. In this paper we focus on a high-level model-based diagnosis of distributed systems like telecommunication networks, power transmission systems, or industrial production processes.

A diagnostic system model is made up of a structural description and a set of component behaviour models. The structure model describes both the system's hierarchical composition in terms of nesting and also dependencies between components such as communication relationships or common power supply. Each component is represented by a non-deterministic input/output automaton. Communication with other components and the environment is implemented as asynchronous sending and receiving of messages through distinct unidirectional channels.

Our work is based on a behaviour reconstruction approach as described by Lamperti, Zanella et.al. [BLPZ99, LZP00, LZ03]. Recorded sensor readings and observed status messages may be considered as a reflection of the system's behaviour and functional

progress. Diagnosis amounts to continuous monitoring of a system and periodical analysis of the relevant output. Behaviour reconstruction traverses all relevant component models to explain observations. A state space is built of valid system behaviour consistent with observations, while incongruous behaviour can be pruned or omitted altogether.

Naturally, the reconstruction process is by far the most resource-consuming part of this approach to model-based diagnosis. In [BLPZ99, LZP00, LZ03] the state space is represented by a product of all involved component automata. Concurrency and asynchronous coupling are essentially represented by interleaving semantics. Because of inevitable combinatorics, this solution suffers from a well-known state explosion problem (see [CGP99], for example).

Here we present an alternative reconstruction algorithm, which makes use of partial order techniques to preserve concurrency and avoid rampant growth of the reconstruction state space. Our goal is to improve the applicability of reconstruction-based diagnosis to real world industrial systems. In future, we intend to apply model-based diagnosis to the domain of railway automation, where status messages from interlockings and control equipment require context-sensitive interpretation to detect causes of (non safety-critical) failures and to guide subsequent restoration.

Section 2 describes basic models and algorithms of automata-based diagnosis. Section 3 then proposes a new model relying on partial-order semantics, for which in Section 4 new algorithms for reconstruction and diagnosis generation are explained. Experimental results are compared in Section 5, also outlining some open questions and persisting problems. Section 6 concludes our work.

2 Diagnosis: Models and Algorithms

The modelling process starts by identifying the requirements for diagnosis. Firstly, the level of detail expressed by the diagnostic model depends on what statements are expected from diagnosis and secondly on what information is available to fill the model with. In general, the behavioural model comprises an abstraction of the functional and the expected faulty behaviour of the component. A system can be hierarchically decomposed into sub-systems or components, which is reflected by the system's structural model. *Components* are model elements at the desired level of detail and abstraction, thus not being subdivided any further.

For each component a model of behaviour must be specified including all known functional states and state transitions. Also a specification of expected faults and malfunctions is required. Knowledge about abnormal behaviour can be obtained through experience or by conducting structured analysis techniques like FMEA, for instance.

An important aspect of a component's behaviour specification is its reaction to communication events or external influences. All control dependencies between components are treated as communications in an abstract way. Each component defines *terminals* (or *ports* in another context) as named endpoints of directed communication channels to otherwise anonymous communication partners. The receipt of a message is an event, which may trigger a transition in the current control state. Furthermore, a component may react by

sending out finitely many messages by itself addressing other terminals.

More formally, a component behaviour model is represented by an input/output automaton:

Automaton $M = (S, T, s_0, R, E)$*, with*
S*: set of control states* s*, initial state* s_0
T*: set of transitions* t*, with* t*:* $s \xrightarrow{q?e/\{r_1!a_1,\ldots,r_n!a_n\}} s'$
$R = R_{in} \cup R_{out}$*: set of terminals, union of input and output terminals* q, r
$E = E_r \cup E_s$*: set of events, union of received trigger events* e *and sent action events* a

The structural model allows to link communication partners to construct larger system models. A *cluster* is composed of sub-clusters, components, and communication channels. A channel connects two component/cluster terminals of matching direction (out→in). Communication is restricted to partners within the cluster. But additionally, clusters can define their own input or output terminals relaying those of members to provide a communication interface to higher level clusters. To realise asynchronous communication, all sent messages are buffered until they are consumed by the recipient. For this purpose, an implicit message queue is assigned to each channel.

Cluster $D = (D_{sub}, C, R, L)$*, with*
D_{sub}*: set of sub-clusters* d
C*: set of components* c
$R = R_{in} \cup R_{out}$*: set of terminals, union of input and output terminals* q, r
L*: set of channels* (r, q)*, with* $r \in c_1.R_{out}$*,* $q \in c_2.R_{in}$ *for some* $c_1, c_2 \in C \cup D_{sub}$

All the information collected at runtime of a system about its status and progress is called the *system observation*. The simplest form of an observation is a single ordered sequence of messages. But a system observation may also be partitioned into multiple sequences, if they are independently provided by observers responsible for different sub-systems. In the latter case it assumed that no further information is available about how to causally relate messages from different sources. It is a sub-task of the reconstruction process to establish these relations.

System observation $Obs(Sys) = \{o_1, \ldots, o_n\}$*, with*
$o_i = \langle (c_j, e_1), \ldots, (c_k, e_n) \rangle$*: message sequence*
(c, e)*: observable message, pair of source component* c *and observed event* $e \in c.E_s$

In the model view of the system, observations are treated as messages sent to the environment. It follows that transitions are either observable or silent. The behavioural model of a component provides implicitly defined 'virtual' terminals for input, output and fault indication. Messages from and to virtual terminals are assigned to transitions just like other communication actions. Sending a message to a virtual fault terminal is a convenient way to mark a transition as faulty and provide a descriptive text (the fault event) at the same time. This feature helps to differentiate between specified normal and expected abnormal system behaviour.

In [LZP00], a diagnostic procedure is proposed that consists of four steps (see Fig. 1). The first step (*definition*) supplies all required input data, that is model, observation and initial

system state. *Task planning*, *behaviour reconstruction*, and *diagnosis generation* form the main three steps, which will be detailed in the following.

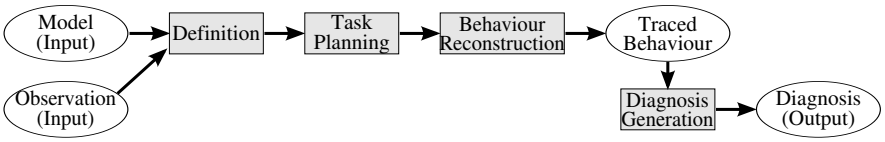


Figure 1: Diagnostic procedure in four steps

Executing diagnosis on a monolithic system model is neither efficient nor practically feasible. One outstanding feature of the diagnosis algorithm by Lamperti, Zanella et al. is that it utilises the hierarchical structure of the model to divide the diagnosis task into sub-tasks. This step is called *task planning*. Starting from individual components, partial diagnostic results are continuously refined (*specialised*) as ever larger clusters of components and their communication relationships are considered.

As introduced, automata-based diagnosis uses a *behaviour reconstruction* approach based on system observation. The work of Lamperti, Zanella et al. defines a reconstruction algorithm (step 3) which incrementally merges (*aggregates*) individual component behaviour into new product automata. During the reconstruction process, new control states are constructed from local control states from all aggregated components. Component transitions consistent with the observation are then copied to those new control states containing their original source state. Basically, concurrent behaviour is mapped onto all interleavings.

The following pseudo-code explains the aggregation of components $c_1, \ldots, c_n$ into a new automaton M_a. The algorithm starts at the combined initial state $g_0 = (s_{10}, \ldots, s_{n0})$ with $s_{i0} \in c_i.S$. The exploration then follows consequently all outgoing local transitions, until already known states are reached or all paths have been visited.

aggregate (components c_i, observation obs) {
  $M_a = (S, T, g_0, R, E)$*;*
  Queue $Q := \langle g_0 \rangle$*;*
  while (Q *not empty)* {
    $g := head(Q)$*;*
    for each local state s *in* g {
      for each outgoing transition t*:* $s \xrightarrow{q?e/\{r_1!a_1,\ldots,r_n!a_n\}} s'$ {
        if (t *activated and consistent with obs)* {
          copy g *to* g'*, update local state* s *with* s'*;*
          copy and update all message queues ($q' := q \setminus e,\ r'_i := r_i + a_i$*);*
          $M_a.T := M_a.T \cup \{t' : g \rightarrow g'\}$*;*
          if ($g' \notin M_a.S$*)* {
            $M_a.S := M_a.S \cup \{g'\}$*;*
            $Q := Q + g'$*;*
  } } } } }
  return M_a*;*
}

A reconstructed *state space* can be viewed as a collection of traces starting at a global initial state and every path containing a valid explanation for what has been observed. Generating a *diagnostic result* from that in the fourth step is simplified to following every path and collecting transitions marked as faulty on the way.

$\Delta = \{\delta_1, \ldots, \delta_m\}$*: diagnostic result, set of possible diagnoses*
$\delta_i = \{(c_j, e_1), \ldots, (c_k, e_n)\}$*: diagnosis, fault set*
(c, e)*: pair of source component* c *and fault event* $e \in c.E_s$

The final diagnostic result Δ is given by a set of alternative diagnoses δ_i, each of them being a set of faults (fault events) together with their source components. Tracking back a faulty transition to its original component provides a solution to both problems of fault localisation and identification.

3 A Partial-Order Model for Diagnosis

A reconstructed state space, represented by a product of automata, shows an excessive growth depending on the number of aggregated components. The two main factors contributing to state explosion are: Firstly, interleaving causally independent behaviour introduces exponential complexity, but adds no relevant information. Secondly, combining all system behaviour into a single product automaton contravenes the general idea of a modular diagnosis. Each component added during the iterative reconstruction process needs to be integrated with all previous components, regardless of the existence of any communication dependencies between them.

To alleviate the well-known state explosion problem, other disciplines dependent on state space representation and exploration have come up with different solutions. *Partial order reduction* techniques [God94, CGP99], originating from the field of Model Checking, keep the interleaving representation, but use the knowledge about concurrency to identify and avoid equivalent paths leading to identical states. *Petri Net unfoldings* [Esp94, HNW99, BFHJ03] take advantage of the partial order semantics of Petri nets. During the unfolding process local behaviour is combined into a global net that keeps the local behaviour separated and introduces event orderings only if induced by communication.

In the next section we present a new reconstruction algorithm for model-based diagnosis employing partial orders. The algorithm relies on an explicit *partial-order model*, which avoids interleavings , but instead directly relates local actions of the involved automata. The overall aim is to impose causal order on communication dependencies and observations but leave concurrent behaviour unordered. Reconstruction shall strictly follow communication links, while keeping individual behaviour models separate. Lastly, reconstruction is just one step in the previously defined diagnostic procedure, which demands compatibility with hierarchical task planning and execution. Hence we need support for partial results and specialisation of these.

We note that the main concern of automata-based diagnosis is to explain a system's observable actions by reconstructing full traces of actions and identifying the faulty ones therein.

From this point of view, local component states play only a minor role in that they merely define which transitions are allowed next, but do not carry any additional information. Also, there exists a strong correspondence between transitions and associated actions because their occurrence is instantaneous, i.e. between triggering (receiving) and sending passes no time. It is therefore sufficient to impose partial ordering directly on transitions rather than individual actions. To support this intention we convert all component automata into an edge graph view (see Fig. 2); Transitions become nodes, and nodes of follow-up transitions are connected by edges. The latter is accomplished for each local state by connecting nodes of all input transitions with those of all output transitions. Initial and final states (if existent) have explicit initial and final transition nodes, respectively.

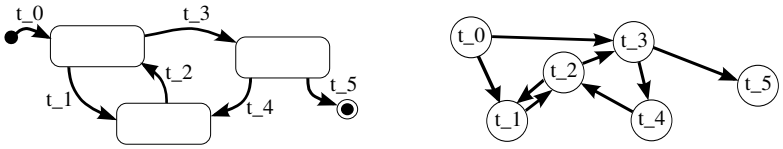


Figure 2: Component automaton to edge graph transformation

The partial-order model builds upon three *relations*: precedence, conflict, and concurrency. Each included element (n_1, n_2) puts two transition nodes into relation. *Precedence* $<$ is a strict partial order that captures sequential behaviour and also causal relationships across components, like sending and receiving of messages, i.e. '$<$' is irreflexive, asymmetric, and transitive. The choice between branching behaviour cannot be expressed by means of ordering. Instead, the *conflict* relation $\#$ mutually excludes transitions which must not appear within the same trace. '$\#$' is irreflexive, but symmetric, and may be not transitive. Moreover, if two tranistions a and b are in conflict, then this is propagated to all successors: $(a \# b) \wedge (b < c) \Rightarrow (a \# c)$ for all a, b, c. Two transitions neither ordered nor conflicting are considered *concurrent*, denoted by $\|$.

To aid the reconstruction process, a fourth relation is introduced. The *validity* relation $\mapsto$ stores pairs of final nodes from different components, that represent valid and consistently terminating behaviour. This helps later to determine matching branches of behaviour from different components in order to derive a final diagnostic result.

A *partial-order model* (POM) $P = (C, Prec, Confl, Val)$ encompasses a set of aggregated components C and precedence, conflict, and validity relations. The POM representation of a basic component, however, starts off with empty relations.

4 Behaviour Reconstruction Using Partial Orders

The partial-order model is accompanied by a new *reconstruction algorithm* that adapts the two main operations: aggregation and specialisation. Following the system's hierarchical structure and starting on the lowest level, the individual behaviour of one or multiple components (or later sub-clusters) is aggregated into a larger cluster. Afterwards, specialisation

restricts the state space to match the observation.

reconstruct (cluster d, observation obs) {
  for each sub-cluster d_{s_i} *in* $d.D_{sub}$
    POM $P_{s_i} := reconstruct(d_{s_i}, obs(d_{s_i}))$*;*
  for each component c_j *in* $d.C$ {
    POM $P_{c_j} := POM(c_j)$*;*
    $specialise(P_{c_j}, obs(c_j))$*;*
  }
  POM $P_d := aggregate(P_{s_*} \cup P_{c_*})$*;*
  $specialise(P_d, obs)$*;*
  return P_d*;*
}

As mentioned before, the component's state spaces shall not be merged into a global state space. Instead, aggregation amounts to copying separate state spaces into a new partial-order model and analysing the additional communication dependencies between them. To avoid re-evaluation of already analysed dependencies, only new links defined in the currently processed cluster are examined. Links connect as per definition only components or sub-clusters on the same level. Hence, links on the current level make up the interface between aggregated components.

The *aggregation* algorithm (see below) takes one ore more POMs as input and produces one new aggregated (cluster) POM as result. After collecting the current set of interface links, relations implied by each communication link are determined (ordering, conflicts etc.). The output POM is constructed by forming the union of all component sets and then combining each of the different relation types together with newly discovered entries.

aggregate (POMs $P_1, \ldots, P_n$*)* {
  POM $R := (\bigcup P_i.C, \bigcup P_i.Prec, \bigcup P_i.Confl, \bigcup P_i.Val)$*;*
  for each link l *in* $interfaceLinks(P_i)$ {
    $(Prec_+, Confl_+, Val_+) :=$ ***order***(l)*;*
    $R.Prec := R.Prec \cup Prec_+$*;*
    $R.Confl := R.Confl \cup Confl_+$*;*
    $R.Val := R.Val \cup Val_+$*;*
    $pruneInvalidBranches(sender(l))$*;*
    $pruneInvalidBranches(receiver(l))$*;*
  }
  return R*;*
}

Ordering is the core mechanism of the reconstruction algorithm that traces asynchronous communication dependencies. Because of the unidirectional nature of a communication channel, clear roles can be assigned to the two connected components acting as sender and receiver of messages. Every sent message must be received somewhere and vice versa, so there exists a causal send-receive-relationship. Also, there is a maximum capacity of the message queue associated with a link, which limits the number of consecutive transmissions.

In order to avoid an overflow, a policy has to be set to wait until some messages have been consumed. Thus, reconstruction has to establish another causal receive-send-relationship to adhere to queue limits. For the time being, queues are fixed to a size of one, which still allows for asynchronous message exchange but makes reconstruction much easier.

During reconstruction, originally cyclic component behaviour is locally unfold into sequential or branching behaviour. The resulting tree structure is bounded in size by the length of the system observation. Still, it is not desirable to expand all branches at once. Our approach features dynamic expansion on exploration, which fully supports focussing on a single communication channel at a time.

The proposed data structure maintains its action centred point of view and continues usage of transition oriented edge graphs. A *dynamic edge graph* (DEG) $D = (c, n_0, N, F)$ refers to a source component c and builds upon a set of nodes N including the initial node n_0. The flow relation F connects transition nodes in such a way that each node has one predecessor and finitely many successors in history.

Starting with the initial node, new nodes are created for each visited transition of the referenced component. Alternative transitions result in new branches. The expansion process continues on all branches until relevant nodes are discovered: nodes sending or receiving through terminals connected with the communication channel under consideration. In case a silent and non-relevant cycle is encountered, expansion is suspended and marked as a repeating pattern. Suspended branches will be unfolded later on, when other communication dependencies are examined, for which this cycle becomes relevant.

```
order (link l) {
  DEG d_s := sender(l); DEG d_r := receiver(l);
  Nodes N_s := {d_s.n_0}; Nodes N_r := {d_r.n_0};
  Prec := ∅; Confl := ∅; Val := ∅;
  while (N_s not empty) {
    /* receive-send-relationship */
    N_s := expandAndSeekNextRelevantNodes(N_s, l);
    for each n_r in N_r
      for each n_s in N_s
        Prec := Prec ∪ {(n_r, n_s)};
    N_r := expandAndSeeNextRelevantNodes(N_r, l);
    /* send-receive-relationship */
    for each n_s in N_s {
      for each n_r in N_r {
        if (terminated consistently) Val := Val ∪ {(n_s, n_r)}
        else if (terminated one-sidedly) Confl := Confl ∪ {(n_s, n_r)}
        else if (valid comm pair) Prec := Prec ∪ {(n_s, n_r)}
        else (comm mismatch) Confl := Confl ∪ {(n_s, n_r)};
    } }
  }
  return (Prec,Confl,Val);
}
```

Through *specialisation* the explored behaviour is limited to what has actually been observed. All paths beginning at the initial state must match the type and order of observed events. Branches that contain inconsistent observable actions are removed from the state space. Furthermore, specialisation has the task to order the occurrences of observable actions across all originating components through precedence. The algorithm to accomplish this is similar to that of ordering of communication dependencies, in that it establishes order between pairs of transitions, which send the observed messages using the observable virtual output-terminal.

Finally, the proposed partial-order model requires a new method for *diagnosis generation* (step 4) because of its strong reliance on data structures. It is the main idea of POMs to avoid an explicit representation of traces, so walking all paths of the state space in order to collect sets of faults is not possible any more. Also, recovering full traces on the basis of previously analysed relations is computationally expensive and inefficient. Fortunately, the consequent separation of individual component behaviour helps once more to simplify the task on hand.

The valid behaviour of each component is stored in a tree structure representing local traces (see Fig. 3a). In preparation of a system diagnosis, local diagnoses are generated first. It is important to note, that the notion of a fault set is not dependent on event order at all. For a local diagnosis it is sufficient to follow all branches (3b) and collect partial fault sets, until all reachable final nodes carry such information (3c).

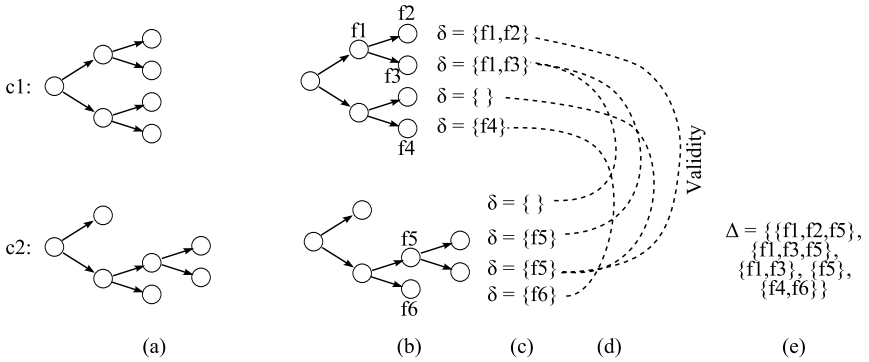


Figure 3: Diagnosis generation (partial-order model)

Now, combining alternative fault sets to generate a comprehensive result resembles a kind of matching problem: finding all valid combinations of exactly one final node taken from each faulty component. All components that do not show any faulty behaviour can already be discarded at this point. In the case of correct system behaviour the resulting diagnosis is empty. Otherwise, the necessary information how to match the remaining fault sets can be deduced from the propagated conflict and validity relations (see Fig. 3d). The diagnostic process ends with the output of Δ, the set of all alternative diagnoses for the entire system (3e).

5 Experimental Evaluation and Open Questions

Unfolding of component behaviour into a tree structure generally carries the risk of massive resource consumption and introduction of even more complexity. The number of nodes in a full binary tree grows exponentially with its depth, whereby the depth is determined by the length of the relevant system observation and the observability of transitions. Model checking [CGP99] intends to exhaustively explore the reachable state space (or a finite prefix thereof) to verify logical properties based on the system model. In contrast, model-based diagnosis focusses on those traces, which include the observation. It therefore only needs to explore a small fraction of the tree structure representing the search space.

Our approach to behaviour reconstruction based on partial orders and dynamic unfolding successfully applies numerous measures to reduce the state space and improve overall efficiency:

- modular diagnosis (components→ clusters→ system), as introduced in [BLPZ99]
- preserve separation of component models
- aggregation follows communication dependencies
- support for true concurrency
- early restriction to observation, pruning of invalid branches of behaviour
- detection and cutting of repeating patterns

Both diagnostic procedures, the original aggregating product automata with interleaving semantics [LZ03] and our approach relating dynamic trees using partial-order semantics, have been implemented in a conceptual prototype written in Java language. The prototype provides all described algorithms (task planning, reconstruction and diagnosis generation) and data structures (automata, partial-order models, and dynamic edge graphs) and is mainly used for testing and evaluation purposes at the moment.

The authors of the original approach have implemented their method in a diagnostic application [CLS$^+$07] too, evaluating complexity of diagnosis on experimental basis. They examine different cases of scalable system models regularly constructed from basic component models. Their main example is taken from the domain of power transmission networks. Power lines are divided into separate segments, each monitored by a dedicated protection system. In case of a short-circuit, breakers at the ends of the affected line segment are opened, effectively disconnecting it from the remaining network. Line protections are dependent in that they provide backup for failing neighbouring protections. The triggering of protections is indicated by status messages, so the diagnosis task concentrates on detection of protection failures by monitoring and analysing backup reactions.

To compare our approach to diagnostic behaviour reconstruction to [CLS$^+$07], we use the same input data, i.e. model descriptions and observations. The referenced experiment ("first class", [CLS$^+$07]) defines a system model consisting of n clusters of type "Protected Line" aligned in a linear sequence.

Table 1 shows a comparison of search space sizes, represented by the count of nodes/edges created during reconstruction, and computation time, both depending on the system size n. The first column is simply copied from [CLS$^+$07] and lists expenses of the original

method. Columns two and three present the results of our prototype, reproducing the results of interleaving and then showing the advantages of partial ordering.[1]

System	Interleaving [CLS$^+$07]			Interleaving Reproduced			Partial Order		
size n	Nodes	Edges	t ($\approx$, sec)	Nodes	Edges	t	Nodes	Edges	t
4	602	857	0.00	236	491	0,1	122	386	0.1
5	1166	1691	0.01	368	922	0,1	153	506	0.1
6	2055	3021	0.02	596	1780	0,1	184	623	0.1
7	3375	5013	0.03	1016	3553	0,3	215	751	0.1
8	5247	7857	0.05	1820	7305	0,9	246	900	0.1
9	7807	11767	0.07	3392	15348	3,6	277	1056	0.1
10	11206	16981	0.13	6500	32682	27	308	1197	0.1
11	15610	23761	0.20	12680	70067	151	339	1349	0.1
...									
15	47121	72621	1.13				463	2034	0.1
20	135286	210361	7.96				618	3060	0.15
25	311451	486951	76.23				773	4325	0.22
30	620741	974141	258.2				928	5707	0.3
...									
100							3098	42019	18.2

Table 1: Comparison of search space sizes (explored nodes and edges)

Both methods based on interleaving show exponential growth in state spaces[2], whereas partial ordering only exhibits polynomial development. Other experiments with partial orders show similarly promising results. However, a formal analysis of complexity of worst and average cases is still under way.

Open Questions

Some open questions remain for future research and the intended application of model-based diagnosis to the domain of railway automation and safety systems. The components in railway automation, i.e. equipment like interlocking, signals, and points are far more complex than those considered in the case study. For high-level process monitoring model-based diagnosis is expected to provide an on-line analysis of malfunctions to guide manual restoration. However, diagnostic information to be provided by the components cannot be defined from the scratch but is defined by legacy or third party components.

Additionally turns out that, when trying to diagnose large systems with complex control dependencies, not the algorithms are the limiting factor but modelling itself. The basic input/output automaton used so far for modelling cannot compactly express dependencies between more than two components. The lacking features to causally join messages from multiple sources lead to another combinational problem. Again, interleaving of otherwise equivalent sequences of messages manifests itself in an exponentially growing number of states needed to create models of controlling components. The primary goal is therefore

[1] To facilitate comparison of statistics, all relations (n_1, n_2) between nodes $n1, n2$ are counted as edges.

[2] nonsignificant deviations due to differences in details of implementation and chosen task planning schemes

to extend the current model's syntax and semantics by new elements and to adapt the new partial order algorithms to all changes without loss of efficiency.

Acknowledgement: We would like to thank Jochen Grühser (Siemens AG, Industry Sector Mobility Division, Braunschweig, Germany) who introduced the topic of rail automation to us and helped detailing the requirements for the application of model-based diagnosis.

6 Conclusion

We have presented a new reconstruction algorithm using partial orders and unfoldings for model-based diagnosis of asynchronously communicating automata modelling distributed systems. Our approach avoids the state explosion problem caused by interleaving, yet fully supporting modularisation of the search process. The experimental evaluation clearly shows the advantages of partial ordering, especially when compared to the original algorithm. Still, some work is required on expressiveness and usability of the models to allow for application of this diagnostic method to technical systems of realistic size and complexity.

References

[BFHJ03] Albert Benveniste, Eric Fabre, Stefan Haar, and Claude Jard. Diagnosis of Asynchronous Discrete-Event Systems: A Net Unfolding Approach. *IEEE Transactions on Automatic Control*, 48(5), 2003.

[BLPZ99] Pietro Baroni, Gianfranco Lamperti, Paolo Pogliano, and Marina Zanella. Diagnosis of large active systems. *Artificial Intelligence*, 110(1):135–183, 1999.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CLS+07] Sergio Cerutti, Gianfranco Lamperti, M. Scaroni, Marina Zanella, and Davide Zanni. A diagnostic environment for automaton networks. *Software Practice and Experience (SP&E)*, 37(4):365–415, 2007.

[Esp94] Javier Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2–3):151–195, 1994.

[God94] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, 1994.

[HNW99] Michaela Huhn, Peter Niebert, and Frank Wallner. Model Checking Logics for Communicating Sequential Agents. In *Foundations of Software Science and Computation Structure, 2nd International Conference (FoSSaCS)*, volume 1578 of *LNCS*, pages 227–242. Springer, 1999.

[LZ03] Gianfranco Lamperti and Marina Zanella. EDEN: An Intelligent Software Environment for Diagnosis of Discrete-Event Systems. *Applied Intelligence*, 18(1):55–77, 2003.

[LZP00] Gianfranco Lamperti, Marina Zanella, and Paolo Pogliano. Diagnosis of Active Systems by Automata-Based Reasoning Techniques. *Applied Intelligence*, 12(3):217–237, 2000.

Structural Analysis of Safety Case Arguments in a Model-based Development Environment

Axel Zechner Michaela Huhn

Institute for Software Systems Engineering
Technische Universität Braunschweig
Braunschweig Germany
{a.zechner|m.huhn}@tu-braunschweig.de

Abstract: The Goal Structuring Notation (GSN) by Kelly [Kel98] facilitates a clear representation of the argument structure in safety cases stipulated by statutory regulations for safety critical systems. We propose a first analysis based on a meta model and OCL constraints that allow to uncover structural incompleteness, inconsistencies, and erroneous instantiations of safety argument patterns. The GSN metamodel is incorporated via a profile into a modelling framework to foster the tight integration between the model-based design and the development of the safety argument. Design decisions regarding safety are reflected in the argumentation and vice versa.

1 Introduction

In many domains, the demonstration of system and software safety in a so-called safety case is requested by the norms (e.g. by the EN50126 [CEN99] for the railway domain) as obligatory premise for certification of safety-critical electronic programmable systems. A safety case is - according to Bishop and Bloomfield - ”a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment” [BB98].

Safety cases are large and complex documents traditionally presented as continuous text with cross-references. As a major improvement, Kelly proposed the Goal Structuring Notation (GSN) [Kel98] that is nowadays well accepted in industry. GSN supports a clear graphical representation of the argument structure, but two issues are only addressed with general hints on best practises and caveats: (1) the argument structure as a piece of reasoning may be incomplete, inconsistent or weakened by other kinds of logical fallacies [Squ06, GKHP06]. (2) The technical evidence backing the logical safety argumentation is missing or weakened. This may result either from significant divergence of the development process or design from the safety arguments or from a collection of design and verification techniques selected for the process that is not sufficient to strongly confirm the required claims. *Assurance based development* by Graydon, Knight et al. [GKS07] propose the methodological integration of the safety case construction into the system development. We take one step forward and suggest the technical integration of the safety

argument structure into a model-based development environment (Sec. 2.2). By extending the meta model with GSN concepts, automated structural analysis like type checking becomes available. Further constraints on structural completeness and consistency of an argument structure can be formalized using the Object Constraint Language (OCL) (see Sec. 3). To complement structural analysis we provide support for checking whether well accepted safety argument patterns are correctly instantiated in a concrete GSN structure (Sec. 3.2). Embedding of the GSN arguments in the modelling framework enables traceability between the arguments and the referenced (sub)system model views (Sec. 3.3). Moreover, this approach allows for exploring the adjacencies of an argument, e.g. the preceding strategies or restricting context information. Tool support provides the technical implementation of the approach (see Sec. 4).

2 The Goal Structuring Notation

The Goal Structuring Notation (GSN) by Kelly [Kel98] aims at the concise presentation of argumentations for safety cases with a graphical notation (see Fig. 1). Like other concepts for logical argumentation GSN follows a concept where a top-level proposition (*Goal*) is decomposed into other propositions until evidence (*Solution*) is available. The key issue of GSN is the clear and structured presentation of safety case arguments with explicitly stating on the underlying constraints and the system context. Typical entities of an argument are identified with typed elements in GSN:

Goal represents a proposition for which evidence is to be provided.

Solution Evidence for a proposition is presented via a Solution element.

Context The Context element constrains the validity of a statement (e.g. to the system, operational environment, etc.).

Strategy Decomposing a proposition (Goal) into subgoals is often ruled by a strategy.

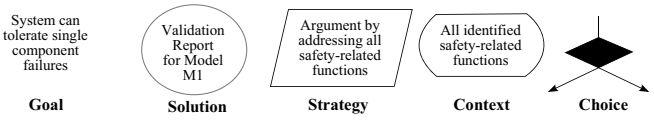


Figure 1: Elements of the Goal Structuring Notation

Relationships between entities are expressed as directed edges. Thus, the GSN elements and their relationships form the so called *goal structure*. This basic set of notational elements is complemented further with elements for *justification* and *assumption*. For premature argumentation, Goals can be marked as *undeveloped*, and alternative lines of argumentation can be denoted with a *choice* element. Hierarchical modelling of arguments and modularization through *packages* helps coping reuse and size of argumentation.

2.1 Existing Tool Support for GSN

We have examined free and commercially available[1] tool support for GSN with focus on integration with model-based development. Additionally, several UML profiles, e.g. EAST-ADL2[2], have "adopted" GSN to their set of modelling elements. Table 1 gives a brief overview of the tool capabilities with a focus on facilities for integration with model-based development and user assistance for computerized assessment and verification.

Name	Integration	Checks	Interchange	Report	Features
ASCE (Adelard Safety Case Editor), Adelard	file links	-	Excel / Access[1]	HTML, DOC	CAE, WBA, FTA schemas
E-Safety Case, Praxis HIS	file links	-	-	-	safety case as website
GSN CaseMaker, ERA Technology	Visio	n.n.	GSML[SFE04]	n.n.	-
ISCaDE (Integrated Safety Case Development Environment), RCM2	DOORS	basic checks	GSML, DOC, etc.[2]	DOC, DB[2]	Hazard Log, Risk Matrix, Requirements
GSN Modeler, Artisan[3]	Artisan	basic checks	XML	n.n.	distributed working
University of York Freeware Visio Add-on	Visio, file links	-	GSML	Word	GSN module support
EAST-ADL2, Atesst	UML 2	-	-	-	-

[1] via external plugin
[2] via DOORS
[3] no evaluation possible, eval. version failed to install

Table 1: Properties of existing GSN tool support

All investigated tools offer a graphical modelling environment for goal structures. GSN elements are presented with their well-known shapes. ASCE is a classical safety engineering environment for systems and focuses on graphical modelling of safety cases and text editing. External evidence can be attached to graph nodes via URLs. Besides GSN, arguments can be modelled using Claim Argument Evidence (CAE). ASCE facilitates further safety related analyses like Why-Because-Analysis and Fault-Tree-Analysis. Additional graphical notations and related functionalities can be provided via schemas and plugins. The Visio-Plugin from York University comes as a rich set of MS Visio stencils and templates with additional editing support via user interface forms. Modelling benefits from the Visio graph drawing environment. It is the only tool which supports hierarchical modelling via GSN modules. External evidence can be attached as file links. According to the publicly available product sheet[3], GSN Case Maker strongly resembles the free visio plugin. ISCaDE already facilitates integration into model-based development of software and systems. The software is based on the DOORS requirements engineering suite and

[1] mostly listed by "GSN User Club" http://www.origin-consulting.com/gsnclub/tools.php
[2] http://www.atesst.org
[3] http://www.cetadvantage.com/website/GSNCaseMaker.aspx

benefits from its strong market position and good integration into third-party development tools. Hazard Log, Risk Matrix, and Requirements Tracing complement this safety case environment. ISCaDE implements computerized evaluation via so-called ”Safety Case Diagram Checks”: root must be a Goal, there must be only one root, and Goal or Strategy must have a Solution. GSN Modeler from Artisan is a rather new graphical modelling tool for developing GSN models offering similar checks like ISCaDE for basic verification. Although evaluation of the evaluation version did not work, it seems to be integrated into the Artisan family of modelling tools. However, the datasheet[4] does not report anything particular about how safety case modelling is integrated into software development or how external evidence can be specified.

E-Safety Case completely differs from the other tools. It is more or less a template or proposal for representing safety cases and related material as a HTML website[CL02], thus not a tool. With EAST-ADL2[The08], GSN modelling within UML is possible but without the familiar graphical notation and it completely lacks further tool support.

All investigated applications allow for nice and easy modelling. The more advanced tool suites offer additional related support for safety engineering like WBA, FTA, Hazard Log, etc . . However, almost all lack of integration into model-based development and support for computerized verification of goal structures. Solely GSN Modeler partially benefits from the use of argument patterns by offering a database. None of the investigated tools facilitates pattern oriented examination [KM97, Kel98, GK08].

2.2 A UML Metamodel and Profile for GSN

Since UML and SysML are widely accepted modelling languages which are well-supported by development tools, we propose a MOF[OMG06] metamodel (see Fig. 2) for GSN. The elements of the metamodel directly relate to the types of argumentation elements (*Goal*,*Solution*, etc.) of the Goal Structuring Notation. All types of elements inherit from *GSNElement*[5], a common ancestor representing shared information properties like a short and a long description. Also for modelling reasons some further elements had to be introduced. E.g., the *Reason* element stands for an semantically implicit meta concept of the both types of argumentation *Justification* and *Assumption*. The same counts for associations and relations between the GSN elements. All relations of elements of a goal structure are semantically bound to the context of the logical argumentation. For the metamodel we wanted to limit the possibility of making links between all kinds of elements to a more restricted and expressive set of associations. Nevertheless, our metamodel does not lack any expressiveness w.r.t. rational argumentations; illegal argumentations are still possible. We will refer to this fact in Sec. 3.

Regretfully, the majority of commercial and open source UML modelling tools are not capable of integrating and mixing external metamodels with their internal representation of UML models. To be able to pursue our goal of integrating GSN into design process

[4] http://www.artisansoftwaretools.com/downloads/support/data-sheets/GSN_Model_NO_CROPS.pdf
[5] not shown for clarity

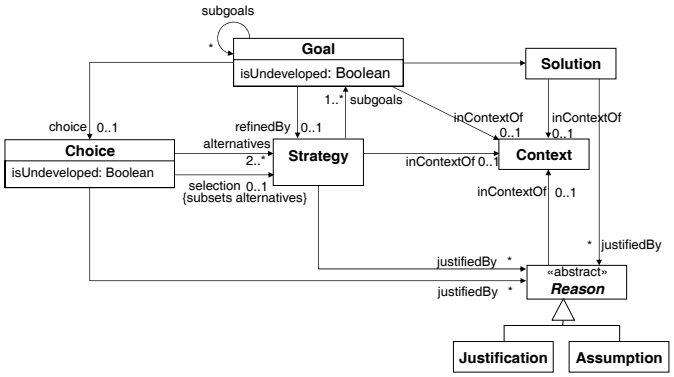


Figure 2: Basic metamodel for the representation of goal structures

and thus into the design tool, the profile mechanism of UML offers an intermediate way of tailoring. From a pragmatical perspective, mapping GSN elements as *Stereotypes* to the *Comment* element fits best since a comment can appear on every kind of UML diagram. Furthermore, we think that mapping GSN elements to *Comments* suits the semantic concept of UML and does not undermine it by ”abusing” arbitrary UML elements. All elements and relations of the GSN metamodel are mapped to *Stereotypes* and element properties to *tagged values*; the short description is mapped to the *Comment*’s body. As an additional feature, the *Context*-Stereotype comprises the ”AnnotatedElements” tagged value[6] for relating to arbitrary kinds of UML elements and a ”URI” field for referring to external evidence documents.

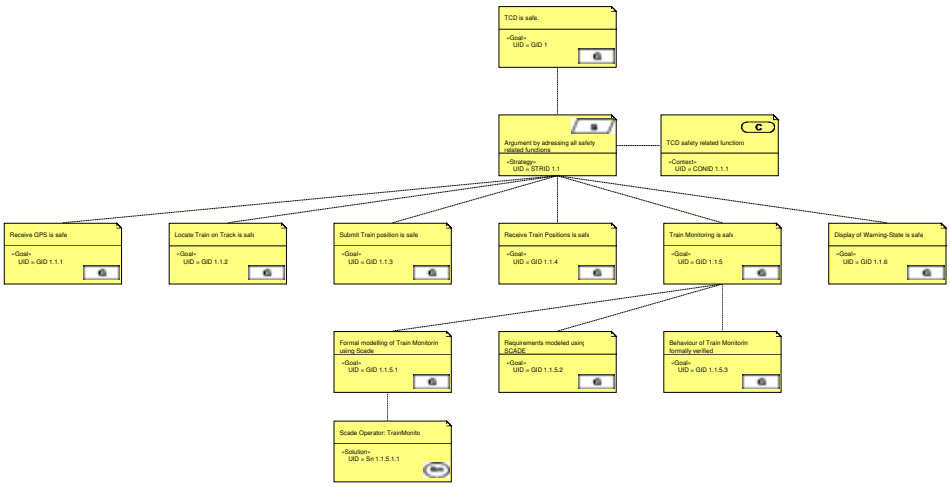


Figure 3: An example goal structure modelled in Papyrus using our GSN profile for UML

[6]equal to associating the *Comment* to a UML element

3 Structural Analysis of Safety Cases

Integration of safety rationale with model-based development improves the effectiveness of both methods. First we present basic computerized analyses of rules and constraints on the structure of arguments using the Object Constraint Language (OCL). Structural analysis is then extended to incorporate evaluation of instances of safety argument pattern. Moreover, the technical integration of safety argumentation into model-based software development facilitates creation, maintenance, and assessment of both.

3.1 Types and Relations

Analysis of the types of and relations between argument entities can be thought of the simplest method for validation of an argumentation where the structural composition is checked for validity while completely neglecting the content and hence the textual semantic of the elements. Although not originally defined, there exists a set of natural, implied rules for the Goal Structuring Notation necessary for a valid argumentation [WMPT96]. Each claim (Goal)

- must either be directly backed by evidence (Solution),
- immediately refined by a set of sub claims forming a decomposition of the higher level claim,
- or, in the case of the GSN, must be refined following a strategy which in turn must decompose into a set of goals.
- No other type of element but a Goal may be the root of a goal structure.

Furthermore, claims, strategies, and evidence must form a directed acyclic graph otherwise we would of course get an argumentation where an ancestor in a hierarchy refers to an antecedent claim as backing. This problem often occurs in pure textual descriptions or can occur in huge and deeply nested hierarchies where arguments are subdivided and scattered over more than one view[7] on the structure. Some more rules arise from examining other arbitrary combinations and relations between the remainder of the GSN elements.

Part of the rules concerning the relations and quantities within relations have already been covered by the construction of the metamodel and profile respectively. This allows us to restrict the possible relations between elements of a concrete model of an argumentation structure to a desired subset already covering the last named rules. Rules about cardinalities of relations and the remaining rules have been implemented using the Object Constraint Language resulting in 19 formulas.

The two most prominent are: *Each Goal must be either refined or backed by evidence.*

```
context Goal inv Goal_OK:
  ((if self.subgoals->size()>0 then 1 else 0 endif)
```

[7] e.g. split over several pages or documents

```
  + (if self.refinedBy->size()=1 then 1 else 0 endif)
  + (if self.choice->size()=1 then 1 else 0 endif)
  = 1) or self.resolvedBy->size()>1
```

The goal structure must not contain a cycle in argumentation.

```
context Goal inv Goal_DAG: not Goal.allInstances
  ->iterate(e;r:Set(Goal)=self.subgoals |
    r->iterate(g:Goal; rs:Set(Goal)=r |
    rs->union(g.subgoals)
      ->union(g.refinedBy.subgoals->flatten())
      ->union(g.choice.alternatives.subgoals->flatten())))
        ->includes(self)
```

Besides those strict syntactic rules one could think of further more heuristic rules for the detection of ill-formed arguments, e.g. chains of single goals possibly indicating a weakness in argumentation, i.e. diversionary arguments or linguistic fallacies. Automated detection of such deficiencies could be achieved in the same manner.

3.2 Argumentation Patterns

The second part of our structural analysis is predicated on the captured knowledge of well proven arguments or argument patterns in the engineering domain. Like design patterns for software[GHJV94],Graydon, Kelly et al. list in [KM97, GK08, Kel98] so-called ”success arguments” or ”argument patterns” which describe strategies for engineering argumentations as abstract fragments of rationales. Such patterns of argumentation show up as result from surveying established safety cases in the engineering field. Thus similar issues are likely to be found in a software safety case which one expects for analysis. Instead of using such patterns during the creation of a rationale, a pattern description also serves well as source for cross-checking an actual argument. Hence, disclosing liabilities and points of weakness is necessarily part of a good pattern description. Examples for argumentation patterns are:

- Functional Decomposition
- Diverse Argument
- Hazard Directed argument
- Compliance
- Use of Existing Evidence
- Formal Method
- Safety Margin

To assure that a pattern is instantiated correctly, it has to be verified that there exists a valid mapping of the present part of the goal structure under investigation to the patterns participants, namely elements of the pattern argumentation: (1) For each participant there must exist at least one corresponding entity in the present goal structure. (2) The relations of a pattern have to be mapped properly to relations of that goal structure. Requiring a one to one mapping of relations seems too restrictive from our point of view. Thus, we propose a weaker condition namely that there exists a path of arguments between arguments being related in the pattern structure.

For an algorithmic testing procedure we will give a formal description of both conditions in the next paragraph.

Refinement

In contrast to [Ede01] who considers not only structural aspects, we describe patterns for argumentation and a refinement mapping on those patterns in terms of acyclic directed graphs and a refinement relationship on these graphs. We denote instances of goal structures as a graph $I = (V_I, E_I)$. The vertices V_I represent the GSN-elements, while relations of a goal structure are mapped to the edges E_I. A safety argument pattern is described as a graph $P = (V_P, E_P)$, too. The vertices V_P represent the participants or roles of a pattern and edges E_P their relations. The reflexive transitive closure of the edge relation E is denoted by $\rightarrow^*$, i.e. a path from v_1 to v_2 is denoted by $v_1 \rightarrow^* v_2$.

We say I *refines* P if there exists a refinement mapping $\Omega \subseteq V_P \times V_I$ with $\forall v \in V_P \exists u \in V_I : (v, u) \in \Omega$, and $\forall (v_1, u_1) \in \Omega$ holds:

$$
\begin{aligned}
&- \forall (v_1, v_2) \in E_P \; \exists (v_2, u_2) \in \Omega : u_1 \rightarrow_I^* u_2 \\
&- \text{and } \forall (v_2, v_1) \in E_P \; \exists (v_2, u_2) \in \Omega : u_2 \rightarrow_I^* u_1.
\end{aligned}
$$

An algorithmic test is easily derived by checking if a refinement Ω fulfils the above formulas which encode reachability.

Extensions to the Metamodel

In order to represent argumentation patterns, we extend the metamodel for GSN (see Fig. 4) and hence the profile with the concepts of pattern specification, pattern instance, pattern roles and refinement mapping. A simple tagging of argumentation elements with role names is not sufficient, because one element of an argumentation could potentially participate in several instances of the same or different patterns, leading to ambiguity. Thus, we introduce new elements to the Goal Structuring Notation similar to the UML concept for *Collaborations* (c.f. [OMG03]).

The specification of a safety argument pattern is realized as a fragment of a goal structure and modelled using the profile. In the metamodel, the *PatternSpecification* represents the anchor element for a pattern specification. A *PatternRole* links a concrete GSN element (Goal, Solution, Strategy etc.), which inherits from *GSNElement*, to its specification tagged with its role name. Each instance of a pattern is indicated by a *PatternInstance* which is related to its corresponding pattern specification. The *InstanceRole* element maps instance entities of a goal structure inheriting from *GSNElement* to its *PatternInstance*. The refinement mapping is realized by relating elements from *InstanceRole*s and *PatternRole*s with identical names.

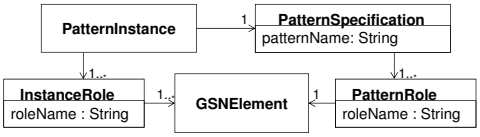


Figure 4: Extensions of the GSN metamodel for pattern

Evaluation of correct instantiation is provided by OCL constraints in the profile. E.g., checking for valid mapping of pattern roles to instances is realized as follows:

```
context PatternInstance inv: self.patternSpecification
  .roleElements->forAll(re|self.instanceElements
    ->exists(ie|re.base_Comment.body=ie.base_Comment.body
      and ie.gSNElement->notEmpty()))
```

3.3 Combination of Safety Argumentation and Embedded Systems Development

In the development of safety critical embedded systems, the safety objectives mutually interferes with the design. On the one hand, safety measures for fault mitigation or protection will result in safety requirements: (1) In parts, safety measures will result in additional functional or behavioural requirements, e.g. fail safe reactions. (2) Design for safety critical systems also puts requirements on the structure or organisation of a software system, e.g. when modularisation of a software system has to obey rules to facilitate automatic verification or when safety-critical functionality has to be separated and deployed independent from the rest counteracting experience from function reuse and object-orientation. On the other hand, design activities like refinement of architecture and design towards implementation can facilitate or complicate creation of safety arguments. In the worst case, a single statement on the implementation level could compromise the entire safety case, e.g. when independence is required, but functions operate on the same data structure. Unfortunately, creating safety cases for software and software development happen to be conducted apart of each other or only loosely coupled.

Ridderhof et al. have demonstrated in [RGD07] how safety argumentation can be integrated into software system design. Based on the fact that safety issues are transformed into safety requirements, their generic argumentation claims that all functions and subsystems fulfil the safety requirements which is backed by requirements tracing in DOORS. However, we think that relating model elements directly to safety arguments is superior because it facilitates comprehension of the impact of safety case design and vice versa.

Co-Developing Software and Safety Case

A typical argument from a software safety case looks as follows:

> *Goal:* Subsystem X has property P because all its subsystems conform to design constraint / decision / requirement C.

When we connect the goal to the model element representing subsystem X for we can capture the constraint, the property which has to be achieved, and also the top-level purpose(s) from which it has been derived. In contrast, a (safety) requirement only comprises information of the affected subsystem and the imposed constraint. Therefore, linking argument and system captures and bundles all relevant safety information for the refinement of subsystem X. On the system model side in turn, safety relevant information carries forward into the model. Safety issues become visible in the composite structure of software components, inheritance hierarchy in object-oriented designs, and associated behaviour

(e.g. state machines). From the safety argument perspective, tight and traceable integration of system development and safety case serves as further supporting evidence of the argumentation for systematic design. Technically this is realized by a set of OCL model queries.

Assessing and Maintaining Software Safety

The assessment of a safety critical system is performed by thorough review and synopsis of safety rationale and system and development artefacts which necessitates collection of safety related data about the system. Maintenance comprises two scenarios: Firstly, a new hazard has been identified and the software has to be modified accordingly. Therefore, the safety case has to be extended and affected functions have to be altered. Secondly, the software shell be extended by functionality or adapted to changes in the runtime environment which presupposes impact analysis on the safety case argumentation. Thus, our approach for integration facilitates both activities, assessment and maintenance, out of the box.

4 Tool Support for Model Analysis

One of our objectives is to leverage and tighten the integration of model-based software development and safety argumentation. The previous sections have already described how this can be achieved by integrating both in the same model description and using existing technology for automated evaluation of goal structures. But besides those necessities, it is inevitable to provide effective instruments to support the relevant activities: (1) modelling of argumentation, (2) concurrent development, maintenance, and assessment of software and argumentation, and (3) model-based evaluation of argumentation structures. We have realized the support as a set of extensions to the Papyrus UML / Eclipse environment.

4.1 GSN Modelling

GSN was designed as a graphical language which benefits also lie in its intuitive visual access. Therefore, we have extended regular modelling by providing a tool palette (Fig. 5(a)) which enables direct creation of stereotyped elements. Additionally, stereotyped GSN comments are displayed with an iconic figure of the respective GSN symbol (see Fig. 3). Thus, Engineers already familiar with GSN can intuitively start modelling.

However, we find the drawing part of the graphical modelling task distracting and time-consuming. As an alternative, goal structures can be created using keyboard-driven dialogues (Fig. 5(c)). A tree widget is used to display the hierarchical relations of the tree-like goal structure. Visualization of goal structures is achieved through automated diagram creation with generated layout.

Pattern descriptions can be modelled the same way and are stored in UML model libraries for later import and reuse. There is no need for additional infrastructure.

The model of a goal structure can be imported and exported to an XML file. The structure

(a) GSN palette (b) Outline view (c) Textual dialogue-driven editor

of our XML schema description facilitates further processing in MS Word which is one of the widest spread engineering and documentation tool. As a positive secondary effect, the XML schema in combination with MS Word also facilitates reconstruction of existing textual argumentations into an input ready to use for our tool environment.

4.2 Integrated Software Development and Assessment

The GSN profile already allows relating safety arguments with entities of the system model. In order to effectively implement concurrent development, maintenance, and assessment of software and argumentation, we have created an additional view called "GSN Outline" (Fig. 5(b)) which key strength is to alleviate synopsis of both worlds by intelligent navigation. It displays context related informations of either GSN or system elements of the currently selected elements. If a GSN stereotyped comment is selected, then all directly related system elements are enlisted as hyperlinks, which when activated open all diagrams containing this element. If e.g. a *Class* is selected on a diagram, then the outline enlists all related GSN elements. Optionally, the outline view can query and display all indirect relations inherited from hierarchy. For classes, that means, that the tool will also display related argumentation elements from ancestors by composition or inheritance in the system model. From the GSN perspective, the view collects all contextual information from *Context* elements up in the hierarchy of argumentation. Argumentation analysis benefits from cross-checking model and implementation and in turn software development becomes aware of safety argumentation and safety related constraints.

4.3 Structure Validation and Reporting

We have customized the verification of OCL constraints in Papyrus UML for more intuitive use with GSN models. All information residing in the model regarding the argumentation can be generated into a HTML report document including results from GSN structural constraint validation and relations from GSN elements to entities of the system.

5 Future Work

Structural analysis of software safety cases is a mandatory first step in safety case assessment, but not sufficient. We are currently working on a method comprising two further analysis steps: (1) We aim at a systematic examination of the conclusiveness of the tree of arguments in software safety cases along the characteristics of the selected development activities and the system's properties. Therefore we will employ a variant of 2-dimensional quality models as suggested by Wagner, Deissenböck et al. [DWP+07] that represents the impact of facts from the system and the process on safety issues addressed in the argument structure. (2) The leafs of an argument structure are solutions for which evidence is provided in terms of technical design artefacts (models and code) or verification or validation reports referring to an artefact. Obviously, also basic tool support for this step like tracing and reporting can be easily provided in a model-based development environment.

6 Conclusion

We have shown how safety case development and model-based development can be beneficially integrated using existing UML modelling technology, minimally extended with computerized support for argument modelling, evaluation and assessment. We proposed structural analyses on the basis of a metamodel extension for GSN argument structures and OCL constraints. Besides type, well-formedness, and completeness checks also the correct use of safety argument pattern can be validated. Our approach is implemented as a UML profile for Papyrus UML. Even in its limitations to syntactic and static analyses, the approach exceeds existing GSN tool support as shown in the tool survey. A first case study in cooperation with Siemens Industry Sector Mobility is under way.

Acknowledgements:
We are grateful to Stefan Gerken from Siemens Industry Sector Mobility for fruitful discussions on software safety case development.

References

[BB98] Peter Bishop and Robin Bloomfield. A Methodology for Safety Case Development. In F. Redmill and T. Anderson, editors, *Safety-Critical Systems Symposium (SAFECOMP)*, pages 194–203. Springer-Verlag, 1998.

[CEN99] CENELEC. EN 50126 – Railway Applications – The Specification and Demonstration of Reliability, Availability, Maintainability, and Safety (RAMS). European Standard., 1999.

[CL02] Trevor Cockram and Ben Lockwood. Electronic Safety Case: Challenges and Opportunities, October 2002.

[DWP+07] Florian Deissenboeck, Stefan Wagner, Markus Pizka, Stefan Teuchert, and Jean-Francois Girard. An Activity-Based Quality Model for Maintainability. In *Proceedings*

of the 23rd International Conference on Software Maintenance (ICSM 2007). IEEE CS Press, 2007.

[Ede01] Amnon H. Eden. Formal Specification of Object-Oriented Design. In *International Conference on Multidisciplinary Design in Engineering CSME-MDE 2001*, pages 21–22, 2001.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[GK08] Patrick Graydon and John Knight. Success Arguments: Establishing Confidence in Software Development. Technical Report CS-2008-10, University of Virginia, Department of Computer Science, July 2008.

[GKHP06] W. S. Greenwell, J. C. Knight, C. M. Holloway, and J. Pease. A Taxonomy of Fallacies in System Safety Arguments. In *International System Safety Conference*, Albuquerque, NM, August 2006.

[GKS07] Patrick J. Graydon, John C. Knight, and Elisabeth A. Strunk. Assurance Based Development of Critical Systems. In *Proc. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks DSN '07*, pages 347–357. IEEE Computer Society, 2007.

[Kel98] Timothy Kelly. *Arguing Safety – A Systemic Approach to Managing Safety Cases*. PhD thesis, University of York, September 1998.

[KM97] T P Kelly and J A McDermid. Safety Case Construction and Reuse Using Patterns. In *16th International Conference on Computer Safety and Reliability (SAFECOMP'97*, pages 55–69. Springer-Verlag, 1997.

[OMG03] OMG Object Management Group. Unified Modeling Language Specification, 2003.

[OMG06] OMG Object Management Group. Meta-Object Facility (MOF) Core Specification, January 2006.

[RGD07] Willem Ridderhof, Hans-Gerhard Gross, and Heiko Doerr. Establishing Evidence for Safety Cases in Automotive Systems - A Case Study. In Francesca Saglietti and Norbert Oster, editors, *26th Intern. Conference on Computer Safety, Reliability, and Security, , SAFECOMP*, number 4680 in LNCS, pages 1–13. Springer, 2007.

[SFE04] Oleksiy Shelest, Saeed Fararooy, and Alan Eardley. Electronic Data Interchange System for Safety Case Management, October 2004.

[Squ06] Matthew John Squair. Issues in the application of software safety standards. In *SCS '05: Proceedings of the 10th Australian workshop on Safety critical systems and software*, pages 13–26, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[The08] The ATESST Consortium. EAST ADL 2.0 Specification. Technical report, The ATESST Consortium, 2008.

[WMPT96] S. P. Wilson, J. A. McDermid, C. H. Pygott, and D. J. Tombs. Assessing complex computer based systems using the Goal Structuring Notation. In *Proc. Second IEEE International Conference on Engineering of Complex Computer Systems*, pages 498–505, 1996.

Reliability Evaluation of Distributed Embedded Systems With UML State Charts and Rare Event Simulation

Armin Zimmermann[1] and Jan Trowitzsch[2]
[1] Technische Universität Ilmenau, System and Software Engineering
armin.zimmermann@tu-ilmenau.de
[2] Max-Planck-Institut für Astronomie, Heidelberg
trowitzsch@mpia.de

Abstract: Safety-critical systems are often controlled by embedded computer systems. Their design is challenging because of the risks connected with the unknown impact of system architecture on non-functional properties such as reliability and real-time capabilities. Model-based evaluation can help to select advantageous design alternatives. This paper proposes the modeling of technical system behavior with UML State Machines extended by stochastic properties (based on the UML Profile for Schedulability, Performance, and Time). The resulting models can be automatically transformed into a stochastic Petri net, for which powerful evaluation methods are available. For models of complex systems, only simulation is possible, which is not feasible for the estimation of safety measures because of the underlying rare events. We propose an evaluation of the resulting models with a recently developed variant of the RESTART rare-event simulation method. A part of the European Train Control System (ETCS) serves as an application example.

1 Introduction

Developing complex technical systems requires adequate methods for the modeling and evaluating of their behavior. Performance and reliability measures can be computed by applying quantitative analysis methods. The *Unified Modeling Language* (UML) [Obj03] is a widely accepted modeling standard in industry. Without extensions, however, UML does not allow modeling and evaluating of properties like timeliness, throughput, or fault tolerance. This paper proposes modeling of technical systems my means of UML State Machines using the *Profile for Schedulability, Performance, and Time* (SPT) [Obj02] to include quantitative system aspects in the model. In order to allow a quantitative evaluation, the resulting model is transformed into a *Stochastic Petri Net*. Performance measures for it can be determined by applying simulation or numerical analysis methods.

Several approaches can be found dealing with quantitative analysis of extended UML diagrams. These often origin from the field of software performance evaluation. Basically two different strategies exist: The direct strategy is to develop and apply an analysis that operates directly on the UML model. The indirect strategy includes the mapping of the UML model into an established performance model such as Stochastic Petri Nets or Queuing Network Models. *Generalized Stochastic Petri Nets* (GSPNs) [AMBC$^+$95] are used by King and Pooley in [KP99, PK99] to represent the behavior of *StateCharts*. Each state is mapped into a place and each state transition becomes a transition in the Petri Net. The resulting sub models are combined using UML collaboration diagrams. Another approach for systematic development of GSPNs is proposed by Merseguer [Mer04] and Bernardi et al. [BDM02]. Extended UML diagrams are translated into labeled GSPN modules, which are merged into a complete model subsequently. In [HSK02] also an extension of UML models with probabilistic choice and stochastic timing is proposed. These indirect approaches have in common that they are limited to exponentially distributed timing, a limitation that is not necessary in our method.

The idea of direct quantitative evaluation of the extended UML model without transforming it into

another model class is followed by Lindemann et al. in [LTK+02]. Deterministic and exponential delays are considered. The resulting stochastic process is a *generalized semi-Markov process* (GSMP), which is numerically analyzable under hard structural constraints, i.e., that activities with non-exponentially distributed delays may only be enabled mutually exclusive in practice.

Simulation is often the only applicable method if the analyzed system becomes too complex. However, standard simulation fails in the case of rare events that are obviously significant in safety-critical systems. The problem is that a simulation only rarely hits an interesting state or event, and will thus need unacceptable run lengths to arrive at a reasonable statistic accuracy for the results of interest. Rare-event simulation is the only tractable method to evaluate such models if they are subject to multiple non-Markovian activity delays and low probabilities of the states under inspection. Several approaches have been investigated in the literature to overcome this problem [GHSZ99, Hei95]. They have the common goal to make the rare event happen more frequently in order to gain more significant samples out of the same number of generated events. We consider the RESTART method [VAVA02a] here because of its robustness and wide range of applicability. An extension of this technique has been propsed in [Zim06], which is used in the context of extended UML state charts in this paper.

A part of the future *European Train Control System* (ETCS) is considered as application example [ZH05]. It provides dynamic track assignment using radio communication when operating at the full implementation level, which is called *moving block operation*. Parameters for reliability and timeliness are included in the existing specifications. However, a detailed investigation of the behavior using a stochastic model to retrieve performance measures has not been carried out during the specification phase.

This paper summarizes and combines results on transforming UML StateCharts with stochastic extensions [TZ06] and rare-event simulation for models with extended performance measures [Zim06]. Additional background on the presented work can be found in [Tro07, Zim07]. The ideas are implemented in our software tool TimeNET [TJZ07, Zim07] as a prototype extension.

The remainder of the paper is organized as follows: Section 2 describes UML State Charts extended by quantitative information using the SPT profile. The transformation of such a model into a stochastic Petri net is explained in the subsequent section. A brief coverage of a rare-event simulation technique used to evaluate models of safety-critical systems is given in Section 4.1. In Section 5, the application example is introduced and results of a sample evaluation are presented.

2 State Charts With Stochastic Extensions

The *Unified Modeling Language* (UML) [Obj03] is a collection of semi-formal models for specifying, visualizing, constructing, and documenting models of technical systems and of software systems. It provides various diagram types allowing the description of different system viewpoints. Static and behavioral aspects, interactions among system components and implementation details are captured. UML is very flexible and customizable because of its extension mechanism. The extension mechanism of UML allows the definition of *profiles*. A profile for a special application domain maps aspects from the domain to elements of the UML metamodel. The *UML Profile for Schedulability, Performance, and Time* (SPT) [Obj02] is an example for such a profile. It enables advanced annotation of quantitative system aspects such as timing and probabilistic information. For this a set of *stereotypes* and *tagged values* is provided. Among the behavioral UML diagrams, State Machines are especially suitable for modeling system behavior. Sequence charts and collaboration diagrams describe single trajectories only. UML State Machines are a variant of Harel's StateCharts [HP98]. They are widely accepted in industry, also because of the availability of suitable software tools.

We do not give a detailed description of UML State Machines and the SPT profile here. In the

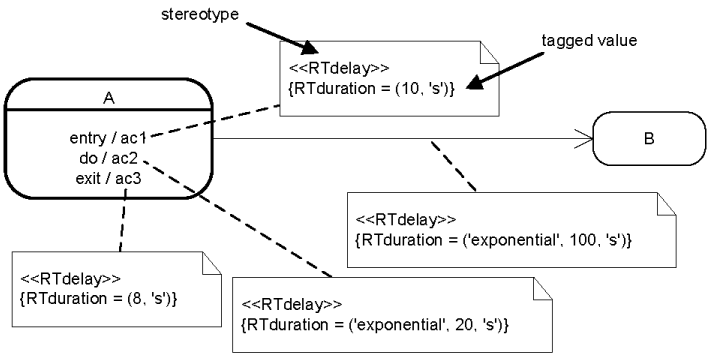


Figure 1: Example of a UML State Machine with SPT annotations

following a short description is given how the relevant quantitative information may be represented within State Machines by means of the SPT profile. For more detailed information we refer to [Tro07, Zim07].

Figure 1 shows an example for a simple annotated UML State Machine. The most important elements within a State Machine are states (`A` and `B` in Fig. 1) as well as transitions. These model state transitions and are depicted as arrows between states. A simple State Machine is always in one state at a time. To each state certain actions may be assigned. In Fig. 1, state `A` has an action `ac1` which is processed during entering of the state (*entry*). Actions `ac2` and `ac3` are processed when the system is in the state (*do*) and upon leaving it (*exit*).

Transitions model a state transition and may depend on guards and generate events. More complex *composite states* are divided into parallel *regions*, suitable for modeling concurrency. Each region specifies its own sequence having a local state. The description of parallel and synchronized processes is easier compared to a classical automata model. Furthermore, State Machines include *pseudo states* which are abstractions that encompass different types of transient vertices such as Fork, Join, or Choice.

Fig. 1 shows examples for the usage of stereotypes and tagged values from the SPT profile. Stereotype `RTdelay` describes the delay of an action, for example 10 seconds for `ac1` in the figure. Tagged values consist of a property name and an assigned value, e.g., `{RTduration=(8,'s')}`. For the timing information it is possible to specify distribution functions for stochastic timing, like `ac2` in Fig. 1. The number value indicates the expectation for the duration as specific parameter for the exponential distribution. The specification of probabilities at transitions can be done using the `PAstep` stereotype and its tagged value `PAprob`.

3 Transformation of UML State Machines Into Petri Nets

In the following we explain our approach for the transformation of UML State Machines into Stochastic Petri Nets aimed at quantitative evaluation. The approach is based on a decomposition of UML State Machines into its basic elements, like states, pseudo states, and transitions. For each element, transformation rules from State Machines into Stochastic Petri Net fragments are specified. Thereby the additional quantitative annotations from the SPT profile are taken into account. Additional timing information are of special interest, such as provided by the *RTdelay* stereotype. The resulting Petri Net fragments are finally composed following the original decomposition.

Petri nets are a model applicable to discrete event systems with synchronized and concurrent pro-

Table 1: Stereotype *RTdelay* - tagged value *RTduration* transformation

Tagged Value	**Petri Net transition**
(8,'s')	deterministic - delay 8 sec
('exponential', 32,'s')	exponential - rate $\lambda = 1/\text{mean}$
('percentile', 80, (5, 's'), 'exponential')	exponential - rate via $F(x) = 1 - e^{-\lambda x}$

cesses. A Petri net is a bipartite graph whose vertices are denoted as *places* and *transitions*. Places may include *tokens*. The distribution of all tokens over the places corresponds with the state of the model. Arcs connect places and transitions and describe the dependency of the active elements (transitions) on tokens in places and their changing due to the transitions firing. For a detailed definition we refer to the extensive literature for this field [Rei85]. Stochastic specifications such as firing times for the transitions were added to Petri Nets in order to enable modeling and evaluation of quantitative system aspects, see [AMBC+95]. In the following, *extended deterministic and stochastic Petri Nets* (eDSPNs) [Ger00] are used.

In the following we briefly explain how timing information is transformed into corresponding Petri Net transitions. Time may be consumed within each state because of the optional internal *entry*, *do*, and *exit* activities. Regardless if the optional activities within a state are specified or not we always follow the temporal and logical order of the activities. If an activity is not specified or if no additional timing information is associated with the activity the corresponding transition in the resulting Petri Net is an immediate transition.

Table 1 presents possible annotations for the *RTdelay* stereotype from the SPT profile and the consequences for the transitions in the resulting Petri Net. Constant times result in deterministic transitions. Exponentially distributed timing results in exponential transitions with the corresponding rate λ. Stochastic timing with a known quantile results also in exponential transitions.

Figure 2 shows an example for the transformation of a simple state transition considering the timing annotations from the SPT profile. The missing *do* activity at state *A* results in the immediate transition *t_do_A*. In many cases these transitions can be automatically deleted from the model later on, in the example leading to a merge of places *A* and *ex_A*. Constant times for the *entry* and *exit* activities result in the corresponding deterministic transitions *t_ent_A* and *t_ex_A* respectively. The state transition from *A* to *B* is assigned by an exponentially distributed delay with the mean value of 100 seconds. The resulting exponential transition *t_trans_A_B* therefore has a rate $\lambda = 1/100$.

Special elements and constructs within UML State Machines include pseudo states, synchronization between regions, and counter variables. Pseudo states are transient vertices with a special seman-

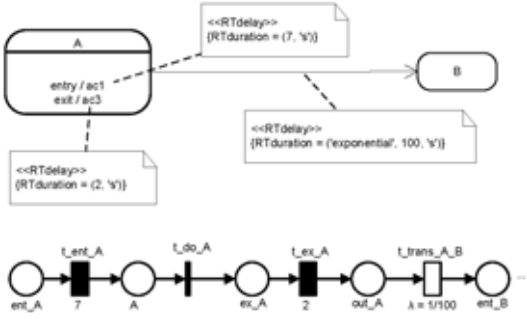


Figure 2: Transformation of a simple state transition

tics which has to be considered during the transformation into a corresponding Petri Net fragment. In [Tro07] transformation rules for *choice*, *join*, *fork*, *junction*, and *initial* pseudo states were introduced among others. The synchronization of regions can be achieved by exchanging events, and results in a corresponding place via which the event is exchanged. An example is used in the application example later. Further details can be found in [Tro07].

4 Rare-Event Simulation for Safety-Critical Embedded Systems

Assume that the goal of a simulation is to estimate the probability $\mathrm{P}\{A\}$ of being in a set of state A in steady state, and that significant samples are generated only rarely due to the model. Let the set of all reachable states of a model be denoted by B_0, and the initial state of the system by σ_0. A standard simulation would require a very long run time until A has been visited sufficiently often to estimate $\mathrm{P}\{A\}$. A is visited more frequently by concentrating on promising paths in the state set.

Formally, define M subsets $B_1 \dots B_M$ of the overall state space B_0 such that $A = B_M$and$B_M \subset B_{M-1} \subset \dots \subset B_1 \subset B_0$ The conditional probabilities $\mathrm{P}\{B_{i+1} \mid B_i\}$ of being in an enclosed set B_{i+1} under the precondition of being in B_i are much easier to estimate than $\mathrm{P}\{A\}$, because every one of them is not rare if the B_i are chosen properly. The measure of interest can then be obtained from the product of the conditionals (obviously $\mathrm{P}\{B_0\} = 1$) as $\mathrm{P}\{A\} = \prod_{i=0}^{M-1} \mathrm{P}\{B_{i+1} \mid B_i\}$.

States visited during a simulation must be mapped to the respective sets B_i. An *importance function* $f_I : B_0 \to \mathbb{R}$ returns a real value for each state $\sigma \in B_0$. A set of *thresholds* (denoted by $Thr_i \in \mathbb{R}, i = 1 \dots M$) divides the range of importance values such that the state set B_i can be obtained for a state[1]: $\forall i \in \{0 \dots M\} : Thr_{i+1} > Thr_i, \sigma \in B_i \iff f_I(\sigma) \geq Thr_i$ We say that the simulation is in a *level* i if the current state σ belongs to $B_i \setminus B_{i+1}$.

An importance splitting simulation measures the conditional probability of reaching a state out of set B_{i+1} after starting in B_i by a Bernoulli trial. If B_{i+1} is hit, the entering state is stored and the simulation trial is split into R_{i+1} trials. The simulation follows each of the trials to see whether B_{i+2} is hit and so on. A trial starting at B_i is canceled after leaving B_i if it did not hit B_{i+1}. Simulation of paths inside B_0 and $B_M = A$ is not changed.

A reduction in computation time results from estimating the conditional probabilities $\mathrm{P}\{B_{i+1} \mid B_i\}$, which are not rare if the sets B_i are selected properly. Even more computational effort is saved by discarding paths that leave a set B_i, and which are therefore deemed unsuccessful. The optimal gain in computation time is achieved by chosing [VAVAGFC94][2] the number of levels $M = -\frac{1}{2}\ln(\mathrm{P}\{A\})$, the conditional probabilites $\mathrm{P}\{B_{i+1} \mid B_i\} = e^{-2}$ and the number of retrials per level as $R_i \approx \frac{1}{\mathrm{P}\{B_{i+1}|B_i\}} = e^2$. It should be noted that the optimal conditional probabilities as well as number of retrials do not depend on the model. Experiences show that the technique works robustly for a wide range of applications [VAVA02a], even if the parameters are not chosen optimally following the rules given in the mentioned papers.

Several variants of RESTART have been considered in the literature [GK98]. We follow a *fixed splitting* and *global step* approach. The first aspect corresponds to the number of trials into which a path is split when it reaches a higher level. The second issue governs the sequence in which the different trials are executed. Global step has the advantage to store fewer intermediate simulation states.

The steady-state value of our example measure $\mathrm{P}\{A\}$ is for a standard simulation given by $\mathrm{P}\{A\} = \lim_{T\to\infty} \frac{1}{T} \int_0^T \mathbf{1}_A(t)\,\mathrm{d}t$ if we denote by $\mathbf{1}_A(t)$ the indicator variable that is either one or zero, de-

[1] We assume $Thr_0 = -\infty$ and $Thr_{M+1} = \infty$ here to simplify notation.

[2] Later results of the same authors [VAVA02b] recommend an alternative setting such that $\mathrm{P}\{B_{i+1} \mid B_i\} = 1/2$, if it is possible to set the thresholds dense enough.

pending on whether the current state of the simulation at time t is in A.

An estimator for this steady-state measure for a RESTART implementation needs correction factors that take into account the splitting. We adopt the method of [TT00], where *weights* ω are maintained during the simulation run, which capture the relative importance of the current path elegantly. This makes the division by $R_0 R_1 \ldots R_{M-1}$ obsolete.

The weights are computed as follows: A simulation run starting from the initial state $\sigma_0 \in (B_0 \setminus B_1)$ has an initial weight of 1, because it is similar to a "normal" simulation run without splitting. Whenever the simulation path currently in level i crosses the border to an upper level u, the path is split into R_u paths, which are simulated subsequently. The weight is obviously divided by R_u upon splitting. Paths leading to a level $< i$ are discarded except for the last one, which is followed further using the stated rules. The weight of the last path is multiplied by R_i when it leaves level i downwards. The weights of the previously discarded paths are thus taken back into consideration, to maintain an overall path probability of one. This procedure is repeated until the required result quality is achieved. This technique has the additional advantage of allowing "jumps of levels" over more than one threshold compared to the original method.

Based on the weight factors, an estimator for the steady-state probability of A is

$$\widehat{\mathrm{P}\{A\}} = \frac{1}{T} \int_0^T \omega(t)\, \mathbf{1}_A(t)\ \mathrm{d}t \tag{1}$$

for a large T, which counts in this context only the time spent in the last path of each split.

4.1 RESTART for Extended Reward Measures

Rare-event simulation approaches concentrate on an estimation of the probability of a rare state set A in transient or steady-state. Others derive the probability of reaching a state or event before another state is hit again. This is, however, a significant restriction in the context of embedded systems evaluation and their respective models. A much wider applicability can be achieved if general quantitative measures are derivable, which heavily depend (perhaps only in some of their terms) on rare events or states.

Instead of estimating $\mathrm{P}\{A\}$, the goal is to obtain an estimation of a reward variable $rvar$. This extension is useful for all performance measures that significantly depend on rewards gained in areas of the state space which are only visited rarely. For simplicity of notation, we restrict ourself to one (possibly complex) measure which is assumed to be analyzed in steady state.

Reward variables $rvar(SProc)$ are functions that return some value of interest from the stochastic process $SProc$ of a stochastic discrete-event model. This process describes the state σ and possibly happening events E at time t, $SProc = \{(\sigma(t), E(t)), t \in \mathbb{R}^{0+}\}$.

Reward variables describe combinations of a (positive) bonus or (negative) penalty associated to elements of the stochastic process. Two types of elements of such a reward variable have been identified in the literature [SM91]. This was based on the basic observation that the stochastic process of a discrete event system remains in a state for some time interval and then changes to another state due to an activity execution, which takes place instantaneously. The natural way of defining a reward variable thus includes rate rewards $rrate(\sigma)$ which are accumulated over time in a state σ, and impulse rewards $rimp(e)$ which are gained instantaneously at the moment of an event $e \in E$.

We first introduce an intermediate function $R_{inst}(t)$. This value can be interpreted as the instantaneous reward gained at a point in time t. It is a generalized function containing a Dirac impulse Δ if

there is at least one impulse reward collected in t.

$$R_{inst}(t) = \underbrace{rrate\big(\sigma(t)\big)}_{\text{rate rewards}} + \underbrace{\Delta \cdot \sum_{e \in E(t)} rimp(e)}_{\text{impulse rewards}} \tag{2}$$

We define the reward variable value in steady-state

$$rvar(SProc) = \lim_{x \to \infty} \frac{1}{x} \int_0^x R_{inst}(t)\,\mathrm{d}t \tag{3}$$

This leads to a simulation estimator $\widehat{rvar}$ in the sense of Equation (1).

$$\widehat{rvar} = \frac{1}{T} \int_0^T \omega(t)\, R_{inst}(t)\ \mathrm{d}t \tag{4}$$

where T is the (sufficiently large) maximum simulation time spent in final paths, and $R_{inst}(t)$ denotes the instantaneous reward gained at time t which is derived by the simulation. $\omega(t)$ denotes the weight as described in the previous section.

It should be noted that the RESTART algorithm stores states with simulation times after a new level has been reached to restart there, which is not visible in Equation 4. Specifically, the algorithm may visit a simulation time t several times with possibly different current states and weights. The equation should thus be read as taking the integral over all paths visited until the global time T (counting only final paths) is reached. This approach requires only a few changes to a standard simulation algorithm, but allows to efficiently estimate reliability measures depending on states which are many orders of magnitude less often visited than states of normal operation.

5 An Application Example

The future *European Train Control System* (ETCS) is being introduced in order to enable fast, efficient, and consistent train traffic across Europe. It is meant to replace the existing national systems. The traditional fixed block structure of the tracks and the release of those track blocks for a train is repealed. In the final implementation (ETCS level 3) a continuous assignment of free track blocks is introduced. Thereby an improvement of the bad track utilization because of the traditional fixed block structure of the tracks ought to be achieved. The traditional track side electromechanical infrastructure is replaced by a radio communication system. The tasks of classical railway control centers are handled by the *Radio Block Centers* (RBC). Every train actively checks its integrity and reports its position to the responsible RBC periodically. Every RBC observes the positions, speeds, and planned routes of the trains within its scope. It assigns to each train free track blocks on which the train can drive safely by transmitting *movement authority* messages to them. This method is called *moving block operation*. For it the reliable and timely data exchange via the radio interface as well as the data processing at the train and the RBC are critical issues for efficient and safe train traffic.

Data exchange between train and RBC is obviously an important issue because otherwise a train could not be informed about the free track blocks along its route. This would make the high speed operation impractical. The connection between trains and RBC is handled wireless via GSM-R (*global system for mobile communications - railway*), a variant of the well-known GSM system for mobile phones [CA00]. The radio communication was specified and designed in detail inside the EIRENE (*European Integrated Railway Radio Enhanced Network*) project. The EURORADIO layer of the communication connection specifies the requirements for the radio communication.

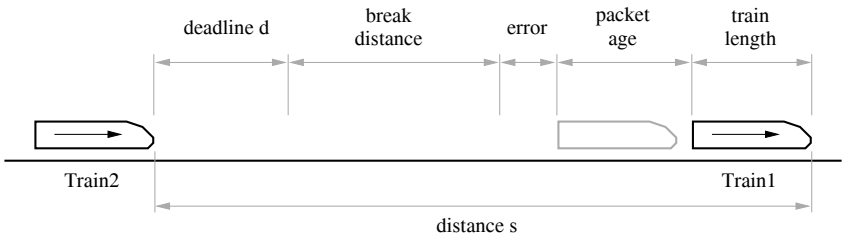


Figure 3: Train Distance and Deadline

In the following the time critical procedure for the determination of the free track section is considered. Thereby the worst-case assumptions from the specification are used to calculate the guaranteed reachable best possible track utilization. First a train checks its integrity. This takes as per specification up to 5 seconds. Afterwards the train transmits the position of the end of the train from the beginning of the integrity check (min safe rear end) to the RBC. This is done periodically every t seconds, with $t \geq 5sec.$ Since the accuracy obviously becomes better if a train sends its position more often we assume in the following $t = 5sec.$

The position message is send via GSM-R to the RBC. This is specified to take between 400 and 500 milliseconds at middle. Processing of data at an RBC takes 500 milliseconds. During this time the movement authority message for the subsequent train is generated. The transmission of this message again takes in middle between 400 and 500 milliseconds.

Communication via GSM-R is not safe. Data packages may be delayed or even get lost. Therefor each train must decide after a certain deadline if a continuation of the drive is no longer safe and an emergency braking has to be initiated. The deadline depends on the driven speed and on the length of the assigned free track section.

We consider two trains *Train 1* and *Train 2*, which drive at the same speed v and directly follow each other. The head-to-head distance is s. Our goal is the calculation of the deadline d for the decision if *Train 2* has to initiate an emergency brake when no new movement authority message arrives. Fig. 3 illustrates this context. The train length (about $410m$ for German high-speed train ICE), the position error of not more than $20m$, and the braking distance (depending on actual speed between $2300m$ and $2800m$ have to be subtracted from the train distance s. We assume in the following the sum of these three parameters as $l = 3000m$.

In the worst-case *Train 1* may have stopped after an integrity check or lost coaches. Because of this the delay a between receiving the message at *Train 2* and the integrity check at *Train 2* also has to be subtracted from the available waiting time. According to the detailed information from ETCS specifications this delay a is between 5 and 9 seconds. The deadline d now can be calculated respectively: $d = \frac{s-l}{v} - a$, whereas $v = 83ms^{-1}$ according to the speed of current ICE trains.

The ability to exchange data packets with position and integrity reports as well as movement authority packets is crucial for the reliable operation of ETCS. In the following we adopt worst-case assumptions based on the requirements from the ETCS specifications, because otherwise there would be no guarantee of a working integrated system. A model of the position report message exchange and emergency braking due to communication problems is developed below. The goal is to analyze the dependency between maximum throughput of trains and reliability measures of the communication system, as well as their impact on economically sound train operation.

Fig. 4 shows the UML State Machine describing the ETCS communication. It includes five parallel regions which are explained in detail subsequently.

The top region models the generation of position/integrity packages at *Train 1*. Such a package is generated every 5 seconds at which an event *TrainSend* is produced. Transmission of data packages

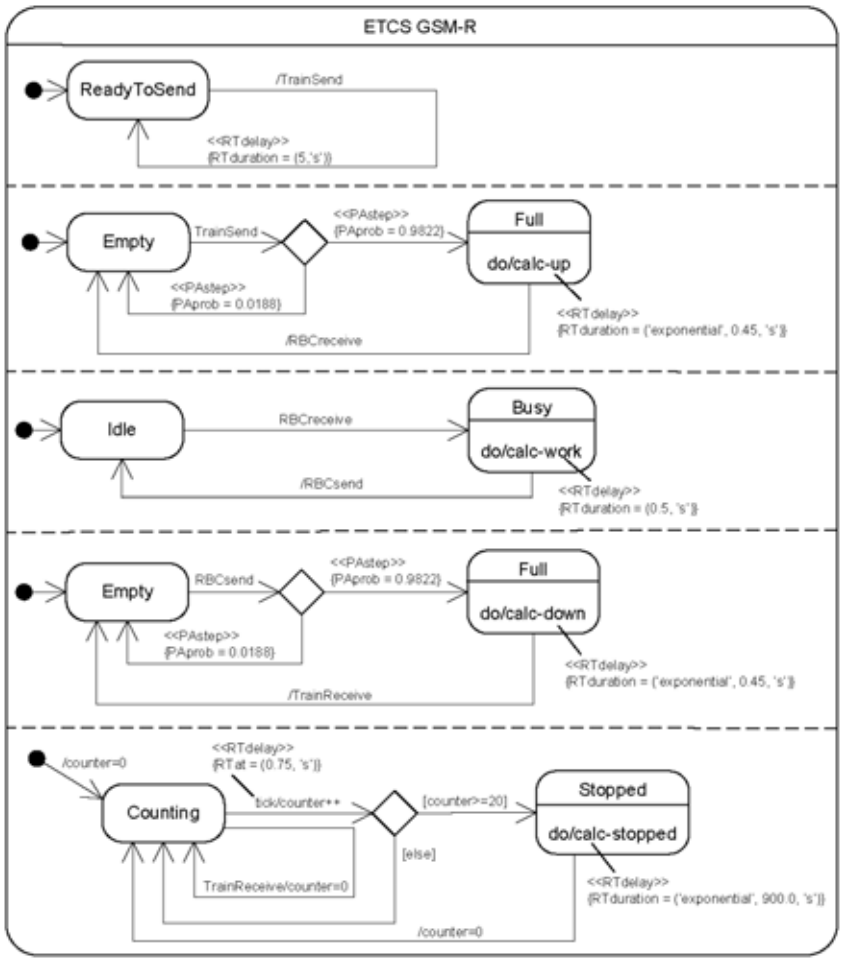


Figure 4: UML State Machine model for ETCS communication

from train to RBC via radio link is described in the region below. The radio link has two possible states *Empty* (no transmission activity) and *Full* (sending of data package). With the occurrence of the *TrainSend* event, a new data package is ready to be sent to the RBC. This data package is correctly send to the RBC with a probability of 98.22%, while with a probability of 1.88% the transmission is incorrect. This is modeled using a *choice* pseudo state and the corresponding *PAprob* annotations at its outgoing transitions. These values result from the bit error rate of 10^{-4} given by the specification and the known package size of 190 bit: $P(error) = 1 - (1 - 10^{-4})^{190} = 1.88\%$. A correct transmission takes 0.45 seconds in total (mean). We do not separate between delays of radio and ISDN backbone transmission here. If the channel is empty again, an event *RCBreceive* is generated during the corresponding transition to state *Empty*. The next region models the behavior at the RBC. With the occurrence of event *RCBreceive* the transition from state *Idle* to state *Busy* is triggered. Processing of a received data package takes 5 seconds. During the subsequent transition to state *Idle* an event *RCBsend* is generated. The region below models the sending of a movement authority message from the RCB to *Train 2*. The only difference to sending from train to RCB are the varying events. Event *RCBsend* activates the transition from state *Empty* to state *Full*. An error may again occur during transmission. An event *TrainReceive* is generated after a correct transmission . The lowest region models observation of the deadline for receiving a new movement authority message at *Train 2*. A counter variable *counter* is used for it. Two states exist: *Counting* and *Stopped*. An

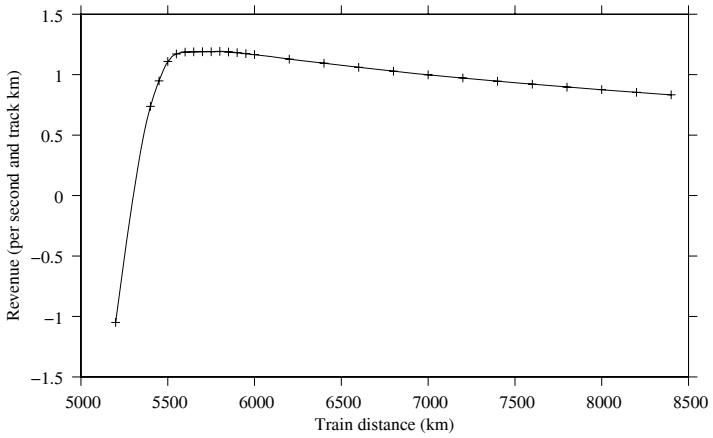


Figure 5: Train revenue versus train distance

event *Tick* is generated every 0.75 seconds, if an exemplary deadline of 15 seconds is considered. With each new *Tick* event *counter* is incremented. If *counter* has reached a value of 20, the train initiates an emergency breaking. For this a delay of 900 seconds in middle is assumed. Afterwards *counter* is set back to 0 and the train starts driving again. If *counter* is smaller than 20, state *Counting* is entered again, waiting for the next *Tick*. A new movement authority message has been received if the region is in state *Counting* and the event *TrainReceive* occurs. In this case *counter* is set back to 0.

The presented model for the ETCS communication is then transformed into a corresponding Stochastic Petri Net as described earlier. Performance and reliability of the model can then be evaluated. The probability for a train being stopped because of a violation of the deadline can, for instance, be obtained by the measure $P(Stop) = P\{\sharp counter >= 20\}$. A steady-state analysis results in the mean probability during operation, i.e., the time a train spends in this undesirable state. This probability is about 10^{-12} for a head-to-head distance of 8 km, for example [].

As an example evaluation we consider the impact of train distance on real-time communication errors, which lead to possible emergency stops and thus losses. We thus define a performance measure to calculate the hypothetical revenue of train operation per second and track kilometer and assume the following settings. In normal operation, 7 Euro are gained per train and second from passenger fares. Whenever an emergency braking occurs, immediate costs of 200.000 Euro are assumed. In addition to that, a stopped train leads to a cost of 600 Euro per second, which also includes stopping of following trains. Assuming a volume-based cost structure of the communication channel, one cent per transferred message in a channel is paid per second. To take track utilization into account, the above influences have to be multiplied by $1000\,\mathrm{m}/distance$ (the number of trains per km). Specification of these values as part of a performance measure is covered in detail in [Zim06].

As we are interested in the average value per model time unit, the performance measure is computed by accumulating the reward over the simulation run and dividing it by the simulation time. Acceptable stop probabilities are naturally very small, and thus a rare-event simulation technique is necessary to successfully derive the measure. A standard simulation would not give reasonable results.

The model of the moving block operation has been evaluated using a prototype implementation in the software tool TimeNET. Thresholds are defined based on the value of the counter variable. The number of thresholds is manually selected in the prototype[3], based on the formulas in [VAVA02a]

[3]TimeNET calculates them based on a prior standard simulation run with limited computation time

and an estimation of the probability of the rare event.

Figure 5 shows the resulting revenue per track kilometer and second, depending on the train head-to-head distance. The optimal distance is 5800m, but the results in the range from 5600 to 5900 do not differ significantly. The two main influences are the rare event of train stops and the reciprocal linear effect of train distance on track utilization. All simulation runs were executed until the rare event was hit at least 1000 times, which required 68 billion events to be simulated in the hardest case of $distance = 8400\,\mathrm{m}$. This took only a few minutes run time on a Pentium Mobile with 1.86 MHz under Windows XP. The same prototype with the RESTART technique switched off is not able to generate any rare event for interesting settings of $distance$ in an acceptable time. The analysis shows a significant impact of communication delays and packet losses on an economical train operation under ETCS level 3.

6 Conclusion

The presented paper describes behavioral modeling of technical with UML State Machines and the subsequent quantitative investigation of such models. For this purpose, extensions from the UML Profile for Schedulability, Performance, and Time are used to introduce the necessary additional information such as delays and occurrence probabilities of actions. A method for the transformation of the resulting models into corresponding Stochastic Petri Nets is proposed. Performance measures can then be calculated by using known Petri Net software tools and algorithms. Reliability measures of safety-critical systems can in many cases only be evaluated by rare-event simulation techniques. The paper shows how the RESTART method can be extended to more general performance measures and applied to distributed embedded systems. As an application example, a part of the communication between trains and Radio Block Centers (RBC) within the future European Train Control System (ETCS) is modeled and evaluated. The methods have been implemented as a prototype extension of the TimeNET tool [Zim07], including a specific graphical editor for UML State Machine models.

References

[AMBC+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Series in parallel computing. John Wiley and Sons, 1995.

[BDM02] S. Bernardi, S. Donatelli, and J. Merseguer. From UML Sequence Diagrams and Statecharts to analysable Petri Net models. In *Proc. of the 3rd Int. Workshop on Software and Performance (WOSP)*, pages 35–45, Rome, Italy, July 2002.

[CA00] Alessandro Coraiola and Marko Antscher. GSM-R network for the high-speed line Rome-Naples. *Signal und Draht*, 92(5):42–45, 2000.

[Ger00] R. German. *Performance Analysis of Communication Systems, Modeling with Non-Markovian Stochastic Petri Nets*. John Wiley and Sons, 2000.

[GHSZ99] Paul Glasserman, Philip Heidelberger, Perwez Shahabuddin, and Tim Zajic. Multilevel Splitting for Estimating Rare Event Probabilities. *Operations Research*, 47:585–600, 1999.

[GK98] Marnix J. Garvels and Dirk P. Kroese. A Comparison of RESTART Implementations. In *Proc. 1998 Winter Simulation Conference*, 1998.

[Hei95] Philip Heidelberger. Fast simulation of rare events in queueing and reliability models. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 5(1):43–85, 1995.

[HP98] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The StateMate Approach*. Wiley, New York, 1998.

[HSK02] R.P. Hopkins, M.J. Smith, and P. King. Two approaches to integrating UML and performance models. In *Proc. of the 3rd Int. Workshop on Software and Performance (WOSP)*, pages 91–92, July 2002.

[KP99] P. King and R. Pooley. Using UML to derive stochastic Petri net models. In *Proceedings of the 15th UK Performance Engineering Workshop*, pages 45–56, Bristol, UK, July 1999.

[LTK+02] C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O.P. Waldhorst. Performance Analysis of Time-enhanced UML Diagrams Based on Stochastic Processes. In *Proc. of the 3rd Workshop on Software and Performance (WOSP)*, pages 25–34, Rome, Italy, 2002.

[Mer04] J. Merseguer. On the use of UML State Machines for Software Performance Evaluation. In *Proc. of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[Obj02] Object Management Group. *UML profile for schedulability, performance, and time*. www.uml.org, March 2002.

[Obj03] Object Management Group. *Unified Modeling Language Specification v.2.0*. www.uml.org, 2003.

[PK99] R. Pooley and P. King. The Unified Modeling Language and Performance Engineering. In *IEE Proceedings - Software*, volume 146(2), March 1999.

[Rei85] W. Reisig. *Petri nets*. Springer Verlag Berlin, 1985.

[SM91] W. H. Sanders and J. F. Meyer. A Unified Approach for Specifying Measures of Performance, Dependability, and Performability. In A. Avizienis and J. Laprie, editors, *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 215–237. Springer Verlag, 1991.

[TJZ07] Jan Trowitzsch, Dan Jerzynek, and Armin Zimmermann. A Toolkit for Performability Evaluation Based on Stochastic UML State Machines. In *Proc. 2nd Int. Conf. on Performance Evaluation Methodologies and Tools (Valuetools 2007)*, Nantes, France, October 2007.

[Tro07] Jan Trowitzsch. *Quantitative Evaluation of UML State Machines Using Stochastic Petri Nets*. Phd thesis, Technische Universität Berlin, 2007.

[TT00] Bruno Tuffin and Kishor S. Trivedi. Implementation of Importance Splitting Techniques in Stochastic Petri Net Package. In Connie U. Smith Boudewijn R. Haverkort, Henrik C. Bohnenkamp, editor, *Computer Performance Evaluation, Modelling Techniques and Tools — 11th Int. Conf., TOOLS 2000*, volume 1786 of *LNCS*, pages 216–pp, Schaumburg, IL, USA, 2000. Springer Verlag.

[TZ06] Jan Trowitzsch and Armin Zimmermann. Using UML State Machines and Petri Nets for the Quantitative Investigation of ETCS. In *Proc. Int. Conf. on Performance Evaluation Methodologies and Tools (VALUETOOLS 2006)*, Pisa, Italy, 2006.

[VAVA02a] Manuel Villén-Altamirano and Jose Villén-Altamirano. Analysis of RESTART Simulation: Theoretical Basis and Sensitivity Study. *European Transactions on Telecommunications*, 13(4):373–385, 2002.

[VAVA02b] Manuel Villén-Altamirano and Jose Villén-Altamirano. Optimality and Robustness of RESTART simulation. In *Proc. 4th Workshop on Rare Event Simulation and Related Combinatorial Optimisation Problems*, Madrid, Spain, April 2002.

[VAVAGFC94] M. Villén-Altamirano, J. Villén-Altamirano, J. Gamo, and F. Fernández-Cuesta. Enhancement of accelerated simulation method RESTART by considering multiple thresholds. In *Proc. 14th Int. Teletraffic Congress*, pages 797–810. Elsevier, 1994.

[ZH05] A. Zimmermann and G. Hommel. Towards Modeling and Evaluation of ETCS Real-Time Communication and Operation. *Journal of Systems and Software*, 77:47–54, 2005.

[Zim06] Armin Zimmermann. Applied RESTART Estimation of General Reward Measures. In *Proc. 6th Int. Workshop on Rare Event Simulation (RESIM 2006)*, pages 196–204, Bamberg, Germany, October 2006.

[Zim07] Armin Zimmermann. *Stochastic Discrete Event Systems*. Springer, Berlin Heidelberg New York, 2007.

From Constraints to Design Space Exploration

Bernhard Schätz, Florian Hölzl, Torbjörn Lundkvist

Institut für Informatik
Technische Universität München
Boltzmannstraße 3, D-85748 Garching
{hoelzlf, schaetz}@in.tum.de

Departement of Information Technologies
Abo Akademi University
Joukahaisenkatu 3–5 A, FIN-20520 Turku
torbjorn.lundkvist@abo.fi

Abstract: Especially in the domain of embedded systems, system development is performed via step-wise design-space exploration, using an incremental addition of design decision. Each development step is characterized by design constraints, limiting the possible solution space. By applying model transformations based on a declarative, relational approach, these constraints can be used to support this exploration of design alternatives. The approach is demonstrated for the (semi-)automatic deployment of logical architectures to hardware platforms.

1 Introduction

The development process – of software systems in general and of embedded system in particular – can be understood as a sequence of design decisions with each step moving from an abstract model – e.g., the description of the logical architecture of a system consisting of communicating (software) components – to a concrete model – e.g., the description of the technical architecture of a system consisting of communicating control units.

Thus obviously, the development process can be understood as a sequence of transformations, each enriching the model under development. Especially in the context of model-driven approaches, model transformation techniques have been developed to support the automatic generation of those transformed models. However, most of those approaches have concentrated mainly on transformations of models to mechanically obtain a single specific transformed model from a given one.

However, the software design process generally involves decision making by the software engineer. Typically, these decision are based on analysis and experience, and cannot be done fully automatic. The decisions are based on abstract models and lead to more concrete, refined models. Often there are multiple possible ways to solve the problem and the developer has to decide which way to go.

This contribution shows how approaches allowing for the definition of loose transformations, i.e., transformations with different possible solutions, can be used to mechanize and interactive and incremental development process.

The remainder of this contribution is organized as follows: Section 2 introduces deployment – i.e., the mapping of logical components and channels to technical units and links – as a typical example of an exploration step in the design process, requiring the evaluation of different variants of such mappings with respect to load restrictions. Furthermore, the relation between such an exploration of the design space and declarative, relational descriptions of model transformations is established. Section 3 gives a short introduction to a representation for conceptual models, enabling the application of those declarative, rule-based transformation specifications to these representations using a Prolog. Section 4 shows how this technique can be used to obtain an interactive and incremental support for the exploration of the deployment possibilites by providing a formalization of the constraints of the solution space. Section 5 finally summarizes the central aspects of the present approach and compares it to related work.

2 Design Space Exploration

In an ideal systems development process, most development activities can be understood as refinement steps from abstract models towards more concrete ones, adding additional information by making design decisions. Besides from rather mechanical activities, obviously these development steps cannot be fully automated, because in general there is not only one possible refinement step to be taken. The development process requires the designer to identify a suitable refinement in case there is more than one possible solution, forcing him to take design decisions, some even being equally optimal. Thus development depends mainly on the developer's abilities and experience of constructing and evaluation these different possibilities.

However, the developer can be supported in his search for the right solution. In the following we show that design space exploration – i.e., the construction of different possible alternatives within a constrained solution space – is suitable problem field to apply (semi-automatic, i.e. interactive) model transformation techniques to guide and support the developer in his decision process.

2.1 Example: Component Deployment

For the demonstration of the approach, we use *component-based deployment* of soft real-time systems as a running example. Component-based deployment, commonly found in the model-based development of embedded software, encompassed the *resource-constrainted mapping* of a set of logical components connected by channels to distributed electronic control units connected by links. The contraints imposed on the mapping from component and channels to control units and links, resp., require the mapping to be *resource-consistent* by enforcing that the (average) load required by a component or channel is less than load provided by a control unit or a link, in case the former are mapped to latter. Components require computational load, while units provide computational load. Chan-

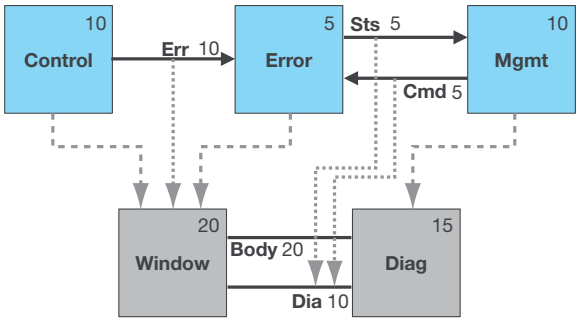


Figure 1: Example: Deployment of Control, Error, and Mgmt

nels require communication load, while links provide communication load.

While a component is always mapped to a unit, a channel is mapped to a link only in case the corresponding components – connected by the channel – reside on different units; it is not mapped to a link if unit-internal communication can be used.

Figure 1 depicts such a deployment for a power window control functionality including error management, mapped to automotive control units for the window movement and diagnosis. The upper half shows the logical components Control, Error, and Mgmt, with channel Err from Control to Error, and channels Sts and Cmd between Error and Mgmt. The required computational and communication load of components and channels is indicated by the adjoined integer numbers.

The lower half shows the control units Window and Diag, connected by links Body and Dia. Again, the adjoined integer numbers indicate the corresponding provided computational and communication load.

Finally, the deployment is shown by arrows from the components and channels to the units and links, resp. Componens Control and Error are mapped to unit Window, while component Mgmt is mapped to Diag. Similarly, channels Sts and Cmd are mapped to link Dia. Since channel Err connects two components mapped to the same unit, it can also be mapped to the same unit.

The mapping is called a *complete deployment* if all components and channels are mapped to units and links. Furthermore, the mapping is called a *consistent deployment* it the required loads of the mapped components and channels do not exceed the provided loads of the units and links they are mapped to. The deployment depicted in Figure 1 is both complete and consistent. E.g., the load required by Control and Error – in total 15 – does not exceed the load of 20 provided by Window. Similarly, the required load of Sts and Cmd – in total 10 – does not exceed the load of 10 provided by Dia.

2.2 Approach

As mentioned above, design space exploration consists of finding a solution from the set of possible designs, respecting some given design constraints. In general, these characteristics of a possible solution in the exploration space can be described in a declarative fashion rather straightforwardly. E.g., as discussed in the previous subsection, a deployment can be easily described as a complete mapping from components to units as well as channels to busses, consistent concerning the provided and required average computation and communication loads.

A mechanised exploration support therefore consists in providing means to automate the systematic search of the design space for those complete and consistent solutions. For that purpose, the declarative description of the design constraints must be turned into an operative version guiding the search process.

To support an effective and efficient process of design space exploration, these operative version should also fulfill additional properties:

- The approach must support an interactive process; i.e., if there are several different solutions to the design problem – e.g., different mappings of components to units – all possible solutions should be presented to the engineer, to support him in making a selection.
- The approach must support an incremental process; i.e., if design constraints are given – e.g., in terms of a partial deployment – all generated solutions should be extensions of these partial solutions.

In [Sch09], an approach is introduced that allows the formalization of (model) transformations by characterizing the properties of a model before and after the transformation in a relational, declarative fashion. By interpreting a model as a structured term, logic programing using Prolog can be used to execute this declarative representation of transformation rules. Since a solution within the design space can be interpreted as a characterization of the model after implementing the corresponding design decision, the exploration process itself can be understood as a transformation step. Of course, this step is generally under-specified and therefore has different possible solutions. Nevertheless, due to the executable interpretation of such a formalization, this approach can be used to automate the search process.

In the remainder, we show how this mechanism can be used to turn a declarative description of design constraints into an automized process supporting the interactive and incremental exploration of the design space. By using a rule-based relational formalization of these constraints, and interpreting them as transformation relation, possible solutions within the design space are generated. Due to the relational style of the formalization and the backtracking mechanism provided by the framework, the different possible solutions can be easily generated. Since the relational approach allows to characterize a *set of possible solutions*, this generation mechanism can be used to incrementally and interactively generate these different possible solutions.

Finally, since the model before the transformation step may already contain elements corresponding to a partial solution, these constraints are directly incorporated in the search process, supporting an incremental approach.

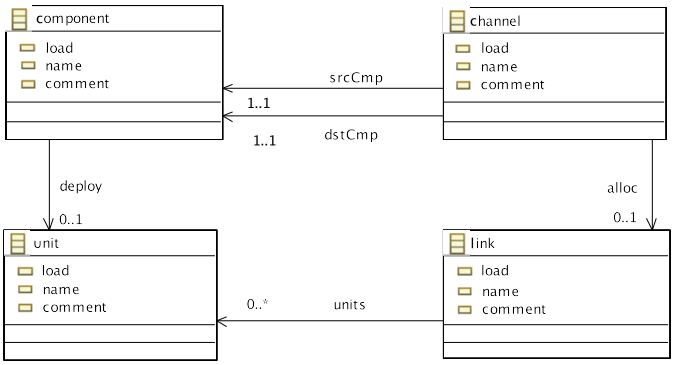


Figure 2: Simplified Conceptual Domain Model for Component Deployment

3 Defining Models, Constraints, and Transformations

As mentioned in the previous, the purpose of the approach presented here is the construction (or rather completion) of descriptions of systems under development – as shown in Figure 1 – to increase the efficiency and quality of the development process. To construct formalized descriptions of a system under development, a 'syntactic vocabulary' – also called *conceptual (domain) model* in [SH99] – is needed. This conceptual model[1] characterizes all possible system models built from the *modeling concepts and their relations* used to construct a description of a system; typically, class diagrams are used to describe them.

Figure 2 shows the conceptual elements and their relations used to describe the logical and technical architectural structure of a system. These concepts are reflected in the techniques used to model a system. In the following subsection, a *formalization of conceptual domain models and conceptual product models* based on relations is given as well as their representation in a declarative fashion using Prolog style.

3.1 Formalization

A *conceptual domain model* provides an interpretation for syntactical descriptions like in Figure 2. Basically, the conceptual domain model defines the primitives used to describe a system: *concepts* characterizing unique entities used to describe a system, with examples in the deployment domain like *component* or *channel* to define the components and channels of the description of the logical architecture of a system; *attributes* characterizing properties, like *name* or *load* to define the name of a component and its required average computational load. Concepts and attributes form the conceptual universe, consisting of

[1] In the context of technologies like the Meta Object Facility, the class diagram-like definition of a conceptual domain model is generally called *meta model*.

a collection of infinite sets of conceptual entities, and a collection of – finite or infinite – sets of attribute values. In case of the deployment, examples for suitable sets of conceptual entities are $CompId = \{comp_1, comp_2, \ldots\}$, and $ChanId = \{chan_1, chan_2, \ldots\}$; typical examples for set of attribute values are $CompName = \{$'Control', 'Error', $\ldots\}$ or $CompLoad = \{0, 1, 2, \ldots\}$.

Based on these primitives, the *conceptual domain model* consists of elements corresponding to objects used to model a system, like *Component*, or *Channel* to define the components and channels of the description of the logical architecture of a system; and relations corresponding to dependencies between the elements, like *srcCmp* or *dstCmp* to define the source or destination component of a channel.

The conceptual domain consists of a collection of element relations between conceptual entities and attribute values, and a collection of (binary) association relations between conceptual entities. In case of the above conceptual domain model for structural descriptions as provided in Figure 2[2], examples for element relations are $Component = CompId \times CompName \times CompLoad$ with values $\{(comp_1,$ 'Control', $5), (comp_2,$ 'Error', $5), \ldots\}$ or $Channel = ChanId \times ChanName \times ChanLoad$ with values $\{(channel_1,$ 'Err', $10),$ $(channel_2,$ 'Sts', $5), \ldots\}$; examples for association relations are $srcCmp = ChanId \times CompId$ with values $\{(channel_1, comp_1), \ldots\}$ or $dstCmp = ChanId \times CompId$ with values $\{(channel_1, comp_2), \ldots\}$. Intuitively, the conceptual domain describes the domain, from which specific instances of the description of an actual system – called conceptual product model in the following – are constructed.

Intuitively, the conceptual domain model is the set of all possible product models that can be constructed within this domain. Thus, each product model is a "sub-model" of the conceptual domain model, with sub-sets of its entities and relations. In order to be a proper product model, such a subset of the conceptual domain model generally must fullfil additional constraints; typical examples are the constraints in meta-models represented as class diagrams. In case of the above conceptual domain model shown in Figure 2, e.g., each channel must have an associated source and destination component in the srcCmp and dstCmp relation.

3.2 Structure of the Model

The transformation framework provides mechanisms for a pure (i.e., side-effect free) declarative, rule-based approach to model transformation. To that end, the framework provides access to EMF Ecore-based models [SBPM07]. As described in Subsection 3.1, formally, a (conceptual) model is a collection of sets of elements (each described as a conceptual entity and its attribute values) and relations (each described as a pair of conceptual entities). To syntactically represent such a model, a Prolog term is used. Since these elements and relations are instances of classes and associations taken from an EMF Ecore model, the structure of the Prolog term – representing an instance of that model – is inferred from the structure of that model. The term comprises the classes and associations,

[2]For simplification purposes, the *Comment* attribute is ignored in the following.

of which the instance of the EMF Ecore model is constructed. It is grouped according to the structure of that model, depending on the package structure of the model and the classes and references of each package. The structure of the model is built using only simple elementary Prolog constructs, namely compound functor terms and list terms.

To access a model, the framework provides construction predicates to deconstruct and reconstruct a term representing a model. Since the structure makes only use of compound functor terms and list terms, only two classes of construction predicates are provided, namely the union operation and the composition operations.

3.2.1 Term Structure of the Model

A *model term* describes the content of an instance of a EMF Ecore model, i.e., the instances of its classes and associations. It consists of a functor – identifying the model – with a classes terms and an associations term as its argument.[3] The *classes term* describes the EClasses of the corresponding package. It is a list of class terms, one for each EClass of the package. Each *class term* consists of a functor – identifying the class - and an elements term. An *elements term* describes the collection of objects instantiating this class, and thus – in turn – is a list of element terms. Note that each elements term comprises only the collection of those objects of this class, which are not instantiations of subclasses of this class; objects instantiating specializations of this class are only contained in the elements terms corresponding to the most specific class. Finally, an *element term* - describing such an instance – consists of a functor – again identifying the class this object belongs to – with an entity identifying the element and attributes as arguments. Each of the attributes are atomic representations of the corresponding values of the attributes of the represented object. The entity is a regular atom, unique for each element term.

Similarly to an elements term, each *associations term* describes the associations, i.e., the instances of the EReferences of the EClasses, for the corresponding package. Again, it is a list of association terms, with each *association term* consisting of a functor – identifying the association - and an relations term, describing the content of the association. The *relations term* is a list of relation terms, each *relation term* consisting of a functor – identifying the relation – and the entity identificators of the related objects. In detail, the Prolog model term has the structure shown in Table 1 in the BNF notation with corresponding *non-terminals* and `terminals`.

The functors of the compound terms are deduced from the EMF Ecore model, which the model term is representing:

- the functor of a ModelTerm corresponds to the name of the EPackage the term is an instance of;
- the functor of a ClassTerm to the name of the EClass the term is an instance of; and finally

[3] Actually, a model term describes the packages of the EMF Ecore model. This aspect of the term representation is skipped here for simplification purposes. A complete description can be found in [Sch09].

ModelTerm	::=	*Functor* (*ClassesTerm*, *AssociationsTerm*)
ClassesTerm	::=	[] | [*ClassTerm* (, *ClassTerm*)*]
ClassTerm	::=	*Functor* (*ElementsTerm*)
ElementsTerm	::=	[] | [*ElementTerm* (, *ElementTerm*)*]
ElementTerm	::=	*Functor* (*Entity*(, *AttributeValue*)*)
Entity	::=	*Atom*
AttributeValue	::=	*Atom*
AssociationsTerm	::=	[] | [*AssociationTerm*(, *AssociationTerm*)*]
AssociationTerm	::=	*Functor* (*RelationsTerm*)
RelationsTerm	::=	[] | [*RelationTerm*(, *RelationTerm*)*]
RelationTerm	::=	*Functor* (*Entity*, *Entity*)

Table 1: The Prolog Structure of a Model Term

- the functor of an AssociationTerm corresponds to the name of the EReference the term in an instance of.

Since EMF – unlike MOF – does not support associations as first-class concepts like EClasses but uses EReferences instead, EReference names are not necessarily unique within a package. Therefore, if present in the ECore model, EAnnotation attributes of EReferences are used as the functors of an AssociationTerm. Similarly, the atoms of the attributes are deduced from the instance of the EMF Ecore model, which the model term is representing:

- the entity atom corresponds to the object identificator of an instance of a EClass, while
- the attribute corresponds to the attribute value of an instance of an EClass.

Currently, while basically also multi-valued attributes can be handled by the formalism, only single-valued attributes like references (including null references), basic types, and enumerations are supported by the implementation.

3.3 Construction Predicates

In a strictly declarative rule-based approach to model-transformation, the transformation is described in terms of a predicate, relating the models before and after the transformation. Therefore, mechanisms are needed in form of predicates to deconstruct a model into its parts as well as to construct a model from its parts. As the structure of the model is defined using only compound functor terms and list terms, only two forms of predicates are needed: union and composition operations.

3.3.1 List Construction

The construction and deconstruction of lists is managed by means of the union predicate `union/3` with template[4]

```
union(?Left,?Right,?All)
```

such that `union(Left,Right,All)` is true if all elements of list `All` are either elements of `Left` or `Right`, and vice versa. Thus, e.g., `union([1,3,5],R,[1,2,3,4,5])` succeeds with `R = [2,4]`.

3.3.2 Compound Construction

Since the compound structures used to build the model instances depend on the actual structure of the EMF Ecore model, only the general schemata used are described. Depending on whether a class/element or association/relation is described, different schemata are used. In both schemata the name of the class or relation is used as the name of the predicate for the compound construction.

Class and Element Compounds The (de)construction of classes/elements is managed by means of class/element predicates of the form `class/2` and `class/N+2` where N is the number of the attributes of the corresponding class, with templates

```
class(?Class,?Elements)
class(?Element,?Entity,?Attribute1,...,?AttributeN)
```

where class is the name of the class and element (de)constructed. Thus, e.g., the class named component in the EMF Ecore model in Figure 2 is represented by the compound constructor `component`. The class predicate is true if `Class` is the list of `Objects`; it is generally used in the form `class(+Class,Objects)` to deconstruct a class into its list of objects, and `class(-Class,+Objects)` to construct a class from a list of objects. Similarly, the element predicate is true if `Element` is an `Entity` with attributes `Attribute1`,...,`AttributeN`; it can be used to deconstruct an element into its entity and attributes via `class(+Element,-Entity,-Attribute1,...,-AttributeN)`, to construct an element from an entity and attributes (e.g. to change attributes of an element) via `class(-Element,+Entity,+Attribute1,...,AttributeN)`, or to construct an element including its entity from the attributes via `class(-Element,-Entity,+Attribute1,...,AttributeN)`. Thus, e.g., `component(Components,[Control,Error,Mgmt])` is used to construct a class `Components` from a list of objects `Control`, `Error`, and `Mgmt`. Similarly, `component(Mgmt,Management,10,"Mgmt","The management component")` is used to construct an element `Mgmt` with entity `Management`, load 10, name `"Mgmt"`, and comment `"The management component"`.

[4] According to standard convention, arbitrary/input/output arguments of predicates are indicated by `?`/`+`/`-`.

Association and Relation Compounds The construction and deconstruction of associations and relations is managed by means of association and relation predicate of the form `association/2` and `association/3` with templates

```
association(?Association,?Relations)
association(?Relation,?Entity1,?Entity2)
```

where association is the name of the association and relation constructed/deconstructed. Thus, e.g., a relation named subComponent in the EMF Ecore model in Figure 2 is represented by the compound constructor `subComponent`. The relation predicate is true if `Association` is the list of `Relations`; it is generally used in the form `association(+Association,-Relations)` to deconstruct an association into its list of relations, and `association(-Association,+Relations)` to construct an association from a list of relations. Similarly, the relation predicate is true if `Relation` associates `Entity1` and `Entity2`; it is used to deconstruct a relation into its associated entities via `association(+Relation,-Entity1,-Entity2)` and to construct a relation between two entities via `association(-Relation,+Entity1,+Entity2)`. E.g., `srcCmp(SrcComps,[ErrCtrl,StsErr,CmdMgmt])` is used to construct the source-component association `SrcComps` from the list of relations `ErrCtrl`, `StsErr`, and `CmdMgmt`. Similarly, `subComponent(ErrCtrl,Err,Control)` is used to construct relation `ErrCtrl` with `Control` being the source-component of `Err`.

4 Exploration by Transformation

As mentioned in Section 2 general, design space exploration is done by imposing additional design restrictions, extending the abstract model by implementation constraints to identify suitable solutions. However, while often it is rather straight-forward to characterize whether a solution is acceptable or consistent, it is more complicated to effectively construct such a solution. Thus, often design space exploration is understood as constructing potential solutions and then checking whether these solutions are acceptable or consistent, often using automatic techniques for the latter step.

In this section we use the example of generating a resource-consistent deployment to illustrate how a characterization of the solution space can be used to effectively perform a mechanized search for a consistent solution in this space. This is achieved by interpreting the declarative characterization of the solution space into an operational description of a possibly ambiguous transformation, allowing to automatically search for suitable solutions within the space.

4.1 Description of Solution Space

In the relational approach, a model is represented as a *single term* using *named compounds* with named constructors as well as *anonymous sets* with union constructor. The model has

```
1  Model = model(Classes,Assocs)
2  Classes = [comp(Components), unit(Units), chan(Channels), link(Links)]
3  Assocs = [srcCmp(SrcComps), dstCmp(DstComps), deploy(Deployments), alloc(Allocations)]
4  Components = [Control, Error, Mgmt], Units = [Window, Diag]
5  Channels = [Err, Sts, Cmd], Links = [Body, Dia]
6  SrcComps =[srcCmp(err,ctrl), srcCmp(sts,error), srcCmp(cmd,error)]
7  DstComps = [dstCmp(err,error), dstCmp(sts,mgmt), dstCmp(cmd,mgmt)]
8  Deployments = [deploy(ctrl,win), deploy(err,win), deploy(mgmt, diag)]
9  Allocations = [alloc(sts,dia), alloc(cmd, dia)]
10 Control = comp(ctrl, "Control",10), Error = comp(error, "Error", 5), Mgmt = comp(mgmt, "Mgmt", 10)
11 Window = unit(win, "Window", 20), Diag = unit(diag, "Diag", 15)
12 Err = chan(err, "Err", 10), Sts = chan(sts, "Stst", 5), Cmd = chan(cmd, "Cmd", 5)
13 Body = link(body, "Body", 20), Dia = link(dia, "Dia", 10)
```

Figure 3: Relational Representation of Power Window Model

a *hierarchic structure*, consisting of *packages* that – in turn – may consist of sub-packages. Each package consists of a set of *classes* and *associations*. Classes and associations consist of sets of *elements* and *relations*, resp., wrapped in compounds. Finally, elements and relations are formalized as compounds of values.

Applied to the running example, the representation of the model in Section 2.1 is shown in Figure 3.[5] As shown in line 1 it only consists of a package named `Model`. Its classes and associations – as shown in lines 2 and 3 – are the sets *Components*, *Units*, *Channels*, and *Links*, as well as *SrcCmp*, *DstCmp*, *Deployments*, and *Allocations*, identified by suitable named compounds (e.g., `comp`, `unit`, `srcCmp`, or `deploy`). These sets – like *Components* and *srcComps* – consist of elements and relations, resp., with themselves are named compounds – like `comp(`*`ctrl`*`,10)` and `srcCmp(`*`err`*`,`*`ctrl`*`)` in lines 10 and 6 – using element identifiers like *ctrl* and *err*.

Using the above formalized representation of a model, the notion of a complete and consistent deployment of components and channels to units and links, resp., can be defined. In the first step, we will only consider the deployment of components to units: A collection Units of units is called *resource-consistent with a collection Comps of components deployed via associations Deploys* if and only if

- Either the sets Comps and Deploys are empty (indicating no to be deployed components)
- Or Units contains a unit Unit with load Load, Comps contains a subset UnitComps, and Deploys contains a subset UnitDeploys such that UnitComps deployed to Unit via UnitDeploys is resource-consistent with Load and the remaining units are resource-consistent with the remaing components deployed via the remaining associations

A collection Comps of components deployed to a unit Unit via associations Deploys is called *resource-consistent with the available (positive) load RestLoad* if and only if

[5] The relational representation uses shortened functors like `comp` for `component`; furthermore, the `comment` attribute is skipped for sake of brevitiy.

```
1  consUnits(Units, [], []).
2  consUnits(Units, Comps, Deploys) :−
3    unit(Unit,Ident,Load,Name), union([Unit], OtherUnits, Units),
4    union(UnitComps, OtherComps, Comps), union(UnitDeploys, OtherDeploys, Deploys),
5    consUnit(Ident, Load, UnitComps, UnitDeploys),
6    consUnits(OtherUnits, OtherComps, OtherDeploys).
7
8  consUnit(Unit, RestLoad, [], []).
9  consUnit(Unit, RestLoad, Comps, Deploys) :−
10   comp(Comp,Ident,CompLoad,Name), union([Comp], OtherComps, Comps),
11   deploy(Deploy,Ident,Unit), union([Deploy], OtherDeploys, Deploys),
12   RestLoad >= CompLoad,
13   consUnit(Unit, RestLoad − CompLoad, Comps, Deploys).
```

Figure 4: Rule-based Formalization of a Deployment

- Either the sets Comps and Deploys are empty (indicating no components to be deployed to units)
- Or Comps contains a component Comp with load CompLoad) deployed via an association Deploy between Comp and Unit in Deploys with sufficient small load (i.e., Load $\geq$ CompLoad) and the remaining collection of components deployed to Unit via the remaining set of associations is resource-consistent with the remaining (positive) load RestLoad $-$ CompLoad

This formalization of a complete and consistent deployment immediately leads to a definition in a declarative manner, using a rule-based relational style. Figure 4 shows the corresponding formalization where `consUnit` checks whether allocation of components to a specific unit respects maximum limit, while `consUnits` checks whether all components are deployed to a unit respects maximum limit. The allocation of channels to links can be performed in a similar fashion, as indicated by the use of relations `consLinks` and `consLink` in Figure 5.[6]

To check the consistent deployments for all units, `consUnits` selects a *Unit* with load *Load* from the set of *Units* (line 3) as well as corresponding subsets *UnitComps* and *UnitDeploys* from the sets *Comps* and *Deploys* of components and deployments (line refDeploy:splitUnitDeploys) – and checks whether all components in *UnitComps* deployed to *Unit* via *UnitDeploys* respect maximum limits via `consUnit` (line 5); unless all components have been deployed (line 1) this is repeated for all remaining components (line 6).

Similarly, to check the consistent deployments for a specific unit, `consUnit` selects a *Comp* with required load *CompLoad* from the set of *Comps* (line 10) as well as the deployment relation mapping *Comp* to the unit under consideration from the set *Deploy* (line refDeploy:consUnitSplitDeploy) – and checks whether the load *Comp* required by the component does not exceed the remaining load provided by the unit (line 12); unless

[6]The definition of relations `consLinks` and `consLink` is skipped for sake of brevity.

```
1  generate(model(PreModel), model(PostModel)) :−
2      model(PreModel,Classes, PreAssocs), model(PostModel, Classes, PostAssocs),
3      comp(Comp,Components), unit(Unit,Units), chan(Chan, Channels), link(Link, Links),
4      union([Comp, Unit, Chans, Links], [], Classes),
5      srcCmp(Src,SrcComps), dstCmp(Dst,DstComps), deploy(PreDep, PreDeploys), alloc(PreAll, PreAllocs)],
6      union([Src, Dst, PreDep, PreAll], [], PreAssocs),
7      union(PreDeploys, AddDeploys, Deploys), consUnits(Units, Comp, Deploy),
8      union(PreAllocs, AddAllocs, Allocs), consLinks(Links, Chans, Allocs, Cons, Deploys),
9      deploy(Deploys,PostDep), alloc(PostAll,Allocs),
10     union([Src, Dst, deploy(Deploys), alloc(Allocs)], [], PostAssocs).
```

Figure 5: Generation of a Deployment

all components have been deployed (line 8) this is repeated for all remaining components (line 13).

The relations `consUnits` and `consUnit` introduced in Figure 4 provide a declarative characterization whether a set `Deploys` of deployment relations between units from `Units` and components from `Comps` is complete and consistent with respect to these components and units. It therefore can be used to *check* whether a given deployment is complete and consistent. However, due to its rule-based declarative character, this definition can also be used to *generate* suitable deployments.

In order to generate the deployment, the above definition has to be embedded in a premodel-postmodel relation, linking a model without deployment to a model extended with a corresponding deployment. Figure 5 shows the embedding, where relation *generate* builds a consistent deployment for a given model resulting in and extended model if possible.

Relation `generate` amends the given model `model(Classes,PreAssocs)` resulting in an extended model `model(Classes,PostAssocs)` by only adding new relations to the pre-model to obtain the post-model and leaving the classes unchanged (line 1). To that end, the deployment relations `PreDeploys` and the allocation relations `PreAllocs` are taken from the `PreAssocs` (line 6) and – after generating a complete deployment if possible via relations `consUnits` and `consLinks` (lines 7 and 8) – added to the `PostAssocs` (line 10).

This `generate` relation can not only be applied to pre-models containing no deployment (as well as allocation) relations; furthermore, the same relation can also be applied to models with an already existing partial deployment (or allocation) in `PreAssocs`, which is extended to a complete deployment (allocation) if possible.

4.2 Execution of Transformation

The approach has been implemented as an Eclipse plugin using the tuProlog engine [DOR05], supporting the transformation of EMF Ecore [SBPM07] models. The implemented plug-in provides tool support both for the definition of transformation and the transformation

execution.

The transformation is provided in form of a transformation wizard, guiding the user through the transformation process of selecting an executing the transformation, and identifying and applying the intended solution. The execution of the transformation itself involves the translation of the pre-model from the EMF to the Prolog representation, the application of the transformation relation, and finally the translation of the post-model from the Prolog to the EMF representation.

As the transformation relation specified in Figures 4 and 5 is executed by the Prolog mechanism, different solutions consistent with this relation can be explored. By repeatedly evaluating the corresponding Prolog term, all possible solutions can be generated and inspected. Once a suitable solution is returned by the Prolog backtracking mechanism, this solution is then transformed to the corresponding EMF form.

5 Conclusion

The development of embedded systems via design-space exploration has been repeatedly defined as an incremental extension of models, e.g., in the Metropolis approach [SLSV00]. However, support for the mechanical exploration of these design options has been sparse. Furthermore, in general, little infra structure is provided to construct such support techniques based on a description of the design constraints, requiring to provide realizations like [AD04] on the level of the tool implementation rather than the conceptual domain level.

Here, an alternative approach is presented, allowing to turn a declarative formalization of the constraints of the design space into an operational mechanism for the generation of solutions for these constraints, interpreting these constraints as transformation relations. For that purpose, a transformation framework – provided as an Eclipse PlugIn [PET09] – is used, supporting the transformation of EMF Ecore models using a declarative relational style. By taking the operational aspects into consideration, the purely relational declarative form of specification can tuned to ensure an efficient execution. Obviously, the rule-based approach allows very general forms of application, using the back-tracking mechanism to explore alternative transformation results.

The purely relational approach combined with the rule-based execution mechanism including backtracking is a necessary pre-requisite to support the design space exploration, lacking in approaches like MOFLON/TGG [KKS07], VIATRA [VP04], or FuJaBa [GGL05]. Also the QVT approach [OMG03] and its respective implementations like ATLAS [JAB+06], F-Logics based transformation [GLR+02], or TEFKAT [LS06] lack its capability to interpret loose characterizations of the resulting model, supporting the exploration of a set of possible solutions. Similar to PROGRES [SWZ99], the approach presented here makes use of a back-tracking mechanism, specifically the one provided by Prolog. However, in contrast to it backtracking is additionally used to produce alternative transformation results not only by automatically searching for an optimized solution, but also be incrementally generating it to allow the user to interactively identify and select the appropriate solution.

References

[AD04] Andrew McCarthy Alan Dearle, Graham Kirby. A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications. Technical report, University of St Andrews, 2004.

[DOR05] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog Integration in tuProlog. *Science of Computer Programming*, 57(2):217–250, 2005.

[GGL05] Lars Grunske, Leif Geiger, and Michael Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In Alan Hartman and David Kreische, editors, *Model Driven Architecture*, volume 3748 of *LNCS*. Springer, 2005.

[GLR+02] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The Missing Link of MDA. In *Graph Transformation*, volume 2505 of *LNCS*, 2002.

[JAB+06] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *OOPSLA '06*, pages 719–720. ACM Press, 2006.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In *ESEC/FSE'07*. ACM Press, 2007.

[LS06] Michael Lawley and Jim Steel. Practical Declarative Model Transformation with Tefkat. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*. Springer, 2006.

[OMG03] OMG. Initial submisison to the MOF 2.0 Q/V/T RFP. Technical Report ad/03-03-27, Object Management Group (OMG), `http://www.omg.org`, 2003.

[PET09] Prolog EMF Transformation Eclipse-PlugIn. http://www4.in.tum.de/ schaetz/PETE, 2009.

[SBPM07] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2007. Second Edition.

[Sch09] Bernhard Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. In Eric Van Wyk Dragan Gasevic, Ralf Laemmel, editor, *Software Language Engineering*, LNCS. Springer, 2009.

[SH99] B. Schätz and F. Huber. Integrating Formal Description Techniques. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99*. Springer Verlag, 1999. LNCS 1709.

[SLSV00] Marco Sgroi, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Formal Models for Embedded System Design. *Design and Test of Computers*, 2000.

[SWZ99] Andy Schurr, Andreas J. Winter, and Albert Zündorf. The PROGRES Approach: Language and Environment. In *Handbook of Graph Grammars and Computing by Graph Transformation*. ,World Scientific, 1999.

[VP04] D. Varro and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *UML 2004*. Springer, 2004. LNCS 3273.

Towards Integrating SysML and AUTOSAR Modeling via Bidirectional Model Synchronization

Holger Giese, Stephan Hildebrandt and Stefan Neumann
[first name].[last name]@hpi.uni-potsdam.de
Hasso Plattner Institute for Software Systems Engineering
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany

Abstract: During the overall development of complex engineering systems different modeling notations are employed. In the domain of automotive systems, for example, SysML models are employed quite early to capture the requirements and basic structuring of the whole system, while AUTOSAR models are employed later to describe the concrete software architecture. Each model helps to address the specific design issue with appropriate notations and at a suitable level of abstraction. However, when we step forward from SysML to the software design with AUTOSAR, the engineers have to ensure that all decisions captured in the SysML model are correctly transferred to the AUTOSAR model. Even worse, when changes occur later on either in the AUTOSAR or SysML model, today the consistency has to be reestablished in a cumbersome manual step. Otherwise the resulting inconsistency can result in failures when integrating the different system parts as captured by the SysML model. In this paper, we present how techniques for the model-driven development domain such as meta-models, consistency rules, and bidirectional model transformations can be employed to automate this task. The concept is exemplified by an experiment done within an industrial project.

1 Introduction

The development of complex engineering systems involves different modeling notations from different disciplines. Taking the domain of automotive systems as an example, SysML (System Modeling Language) [Sys08] models are employed quite early to capture the requirements and basic structuring of the whole system and AUTOSAR (Automotive Open System ARchitecture)[1] models are used later in the development process to describe the concrete software architecture and its deployment. Using these different model helps to address each specific design issue with an appropriate notation and at a suitable level of abstraction.

When going from the system design with SysML to the software design stage with AUTOSAR, today, the engineers have to ensure manually that all decisions captured in the SysML model are correctly transferred to the AUTOSAR model. When changes occur later on either in the AUTOSAR or SysML model, the situation is even worse: The consistency has to be reestablished in a cumbersome manual step that inspects both models and transfers all detected changes. Otherwise, the integration of the different system parts as captured by the SysML model and refined in the AUTOSAR model may fail.

Model-Driven Engineering and model transformation are a promising direction to approach this problem (cf. [Win07]). Models are used to describe the system under development from different viewpoints and on different levels of abstraction. The use of different kinds of models leads to the problem of keeping those models consistent to each other. At this point, model transformation systems play a central role.

Triple graph grammars (TGGs) are a formalism to declaratively describe correspondence relationships between two types of models. They were introduced in [Sch94]. A TGG based transformation system can perform model transformations using this declarative transformation specification.

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

Figure 1: A simple meta model to describe components, ports and connectors

In several context different variants of TGGs have been employed for model synchronization such as the integration of SysML models with Modelica simulation models [JPB08], keeping models from the domain of chemical engineering consistent [BHLW07] and transformations from SDL models to UML models and vice versa [BGN+04].

This paper reports about a project with industry in which we investigated how the SysML tool TOPCASED and the AUTOSAR tool SystemDesk can be integrated. We use techniques from the model-driven development domain such as meta-models, consistency rules and bidirectional model transformation resp. model synchronization to automate the integration task. The model transformation permits to automatically derive initial AUTOSAR models from SysML models and to reestablish consistency between both models in case of changes in one of them. More specifically, our model synchronization approach [GW09] based on triple graph grammars (TGG) [Sch94] has been employed to synchronize both models such that changes within one of them are automatically transferred to the other one.

The automatic synchronization of models not only reduces the costs and time required for the transition from the SysML to the AUTOSAR model. It also reduces the cost and time to reestablish consistency in case of changes in either model. In addition, the automated synchronization is less error prone than manual labor employed today and enables to employ iterative and more flexible development processes as the costs for iterations or changes are dramatically reduced.

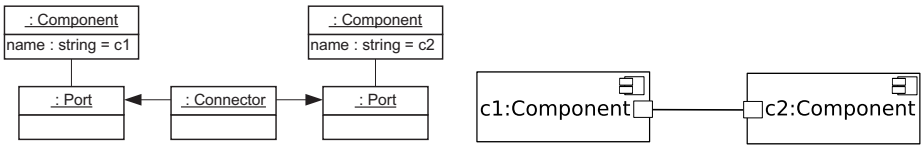
The structure of the paper is as follows: The employed concepts from MDE are outlined in Section 2. The considered modeling notations SysML and AUTOSAR are introduced in Section 3 and the model synchronization between both are presented in Section 4. The approach and its support for model synchronization are presented in Section 5 before we discuss its suitability for several typical usage scenarios, such as the initial transfer of information or change propagation, in Section 6. The paper closes with a final conclusion and an outlook on planned future work.

2 Model-Driven Engineering

At the core of the *Model-Driven Engineering* (MDE) approach, models build the basis for the development of systems. Different kinds of models are used to describe the system under development from different viewpoints and on different levels of abstraction. While these models are all related to each other, they need to be kept in sync after modifications. For this purpose, model transformation and synchronization systems can be used that create a new target model from an existing source model (transformation) or modify an existing target model to make it consistent to a source model (model synchronization). More details on model transformation will be presented later in Section 4.

2.1 Meta Models

In MDE, general purpose modeling languages as well as *Domain-Specific Languages* (DSLs) are commonly used to describe problems specific to the domain where the system is to be deployed. Such languages can be textual or visual languages. Commonly, meta models are used to describe the abstract syntax of such languages. A meta model defines all the elements that can be used in a valid model, as well as the relationships between them. Figure 1 shows a simple meta model for the description of components, ports and connectors. Figure 2 shows an example instance model in abstract and concrete syntax. The abstract syntax shows all elements of the model as objects in an object diagram. The concrete syntax uses a specific graphical notation.



(a) Abstract Syntax (b) Concrete Syntax

Figure 2: An example model conform to the meta model in Figure 1

2.2 Constraints on Models

Usually, UML Class Diagrams are used to describe the structure of a meta model. However, there are some properties that cannot be expressed using class diagrams but which must be fulfilled by valid instances of a meta model. An example is a constraint on the value an attribute may take. For this purpose, the *Object Constraint Language* (OCL) can be used to express such additional constraints. In Figure 1, an OCL constraint is used to describe, that the name of a component must be shorter than twenty characters. These constraints can be evaluated on instances of the meta model to check if these are indeed valid instances.

2.3 Profiles and Stereotypes

Instead of using meta models to define a modeling language, the UML can be adapted by profiles and stereotypes. Stereotypes can be used to add new attributes to existing UML elements. They can also define constraints that must be fulfilled by instances of the stereotyped meta elements. A profile contains a set of stereotypes and can be applied to a UML model. The following chapter explains the use of stereotypes for SysML models.

3 Modeling

There exist several suitable modeling languages and notations for the development of complex systems (e. g., for embedded automotive systems), which focus on different aspects or views. In this paper, we have a look into two different languages that are used within a particular development thread going from the system engineering (including requirements as well as the HW and SW structure) down to software engineering. The application domain considered here is the development of automotive embedded systems. SysML is used to analyze and design the overall system architecture and the AUTOSAR framework is used to specify the SW architecture in more detail. AUTOSAR is a DSL, which focuses on the development of Electronic Control Systems. This is only one aspect of the overall system engineering process supported by the SysML modeling language.

3.1 SysML

A widely-used language for system engineering is SysML (System Modeling Language), which is currently available in version 1.1 (see [Sys08]). SysML supports the desing and analysis of complex systems including HW, SW, processes and more. SysML reuses a subset of the UML and adds some additional parts (e. g., the Requirement- and Parametric-Diagram) to facilitate the engineering process by providing several improvements compared to the UML concerning system engineering. The UML itself tends to be more software centric while the topic of SysML is clearly set to the analysis and design of complex systems (not only SW).

In SysML, system blocks are used to specify the structure of the system[2]. For this purpose the UML element *Class* is extended by the stereotype *block*. A block describes a logical or physical part of the system (e.g., SW or HW). Multiple of these blocks can be used for representing the structure of a system. An example for the additional capabilities of SysML is the possibility to model the flow of objects between different system elements (which are specified in form of

[2] A block describes a part of the structure of a interconnected system.

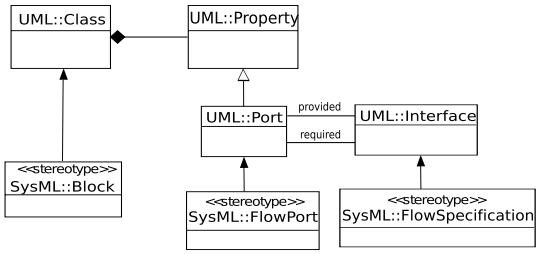


Figure 3: Extract of the SysML metamodel

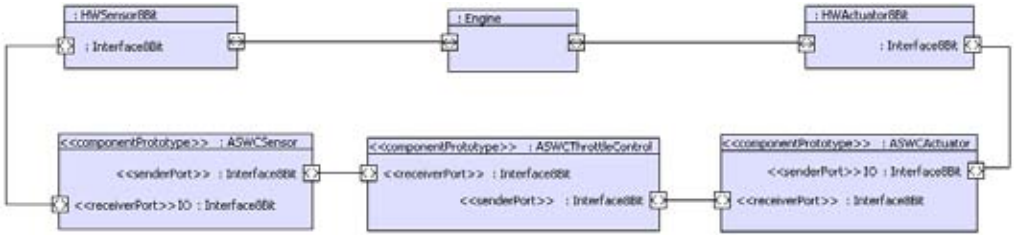


Figure 4: Application example of an SysML model created in Topcased

SysML blocks) by the usage of *flow ports*. A *flow port* is a stereotype for the UML element *Port* and allows the modeling of an object flow between SysML blocks. For the specification of objects and data, which flow over a flow port the stereotype *flow specification* is applied to the UML element *Interface* in SysML. The SysML meta model describing blocks, flow ports and flow specifications is shown in Figure 3.

When analyzing and designing automotive systems, the HW/SW-structure can be described using SysML blocks, ports (e. g., flow ports) and appropriate interfaces (e. g., flow specifications). In this paper, we use a simplified version of the structural constituents taken from an application example of an engine-fuel control system consisting of actuators and sensors for the throttle position and the control software. The control software evaluates the sensor values, computes appropriate throttle position values and sends them to the actuator of the throttle.

The system structure including HW and SW parts is modeled using the tool TOPCASED[3] and the resulting SysML model of the engine fuel control system is shown in Figure 4. The example consists of six different types of blocks, three of them represent hardware parts like the engine, a HW actuator and a HW sensor for setting and measuring the throttle position of the engine. The HW sensor (*HWSensor8Bit*) is connected to a SW block (*ASWCSensor*), which reads in data from the HW (e. g., by using driver functionality) and sends these measured values to a SW block, which realizes the control functionality (*ASWCThrottleControl*) and computes an output signal. This output signal is send to a SW block (*HWActuator*), which realizes the access to the HW actuator, which is represented through the block *HWActuator8Bit*. The *HWActuator* interacts with the representation of the physical engine.

When such a system is designed several restrictions have to be considered concerning the used HW sensor blocks in combination with the software blocks. A typical restriction is that a connector could only connect ports, which implement the same interface. In the shown example, e.g., the flow ports of the blocks *ASWCSensor* and *HWSensor8Bit* over which these two blocks are connected, have to implement the same interface. Such a constraint can be expressed in form of the following OCL constraint for the type connector:

[3]http://www.topcased.org/

```
context Connector inv:
self.end->forAll(e:self.end->get(0).role.type == e.role.type)
```

Only three of the blocks (*ASWCThrottleControl*, *ASWCSensor* and *ASWCActuator*) described above are relevant for the SW architecture. We use stereotypes to be able to identify the definition and the usage of SW blocks like described in Section 2. In our implementation, stereotypes are defined for identifying, e.g., the definition of SW blocks (*atomicSoftwareComponent*) as well as for the usage of the defined SW blocks (*componentPrototype*) like shown in Figure 4. In the following section, we show how these constituents can be represented in a DSL, which focuses on the development of automotive software systems.

3.2 AUTOSAR

AUTOSAR (Automotive Open System ARchitecture) is a framework for the development of complex electronic automotive systems. The purpose of AUTOSAR is to improve the development process for ECUs (Electronic Control Units) and whole systems by defining standards for the system and software architecture. The AUTOSAR standard[4] defines a meta model, which describes a DSL for the development of automotive embedded systems. This meta model is described in [AUT07] in form of an UML profile. We use a stand-alone meta model for AUTOSAR, which is realized accordingly.

As defined by the AUTOSAR meta model the software architecture is build of Components (e. g., *AtomicSoftwareComponents* (ASWC)). These ASWC are derived from the type *ComponentType* and can communicate using two different categories of ports, required and provided ports (represented through *RPortPrototype* and *PPortPrototype*). Both types are derived from the same abstract class *PortPrototype*. An *RPortPrototype* only uses data or events, which are provided by other ports of type *PPortPrototype*. A port of type *RPortPrototype* or *PPortPrototype* can implement an interface of type *PortInterface*. This *PortInterface* is refined by *ClientServerInterface* and *SenderReceiverInterface*. The AUTOSAR meta model for SWCs and for the different types of ports is shown in Figure 5.

The SW blocks (*ASWCSensor*, *ASWCActuator* and *ASWCThrottleControl*) defined within the SysML model described above can also be specified within an AUTOSAR model. The blocks shown in Figure 4 can also be described using ASWCs, ports and interfaces, which are defined within the extract of the AUTOSAR meta model shown in Figure 5. Figure 6 shows the same SWCs modeled with the tool SystemDesk[5]

In case of the SysML example, the SW blocks, ports and connectors can be described directly within such an AUTOSAR model in form of ASWCs. In case of the blocks describing HW, such a mapping is currently not realized in our system. Therefore, the blocks, ports and connectors concerning HW in the SysML model do not exist in the AUTOSAR model. Also the connectors, which exist in the SysML model between the ports of a SW block and a HW block are not transformed to AUTOSAR. In the next project phases, also these HW constituents will be considered. The AUTOSAR model described in this specific case is a subset of the elements existing within the SysML model. One possibility to derive the AUTOSAR model shown in Figure 6 from the SysML model is to manually transfer the relevant parts. Such a manual activity is expensive and error-prone. Furthermore, to manually keep both models consistent when changes occur is even more difficult. Another possibility is to use techniques, which allow to automatically derive one model from another or even to synchronize two existing models. Subsequently we describe a technique, which supports model transformation as well as model synchronization.

4 Model Synchronization

Model transformation systems can be used to transform one model into another model using a set of transformation rules. These rules are defined on the meta models of the source and

[4] Information can be found at: http://www.autosar.org

[5] http://www.dspace.com/ww/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm

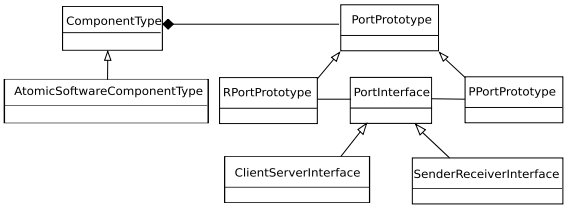


Figure 5: Extract of the AUTOSAR meta model

Figure 6: AUTOSAR model derived from the SysML model

target models of the transformation. They describe, what pattern of target model elements has to be created if the source model contains a certain element pattern. The model transformation system analyzes the source model and creates a target model according to the transformation rules. Examples of model transformation systems are ATL[JABK08], VIATRA[CHM+02] and systems based on *Triple Graph Grammars*[GW09] (TGG), as well as the QVT standard[OMG], which describes a language for expressing model transformation rules but does not provide an implementation.

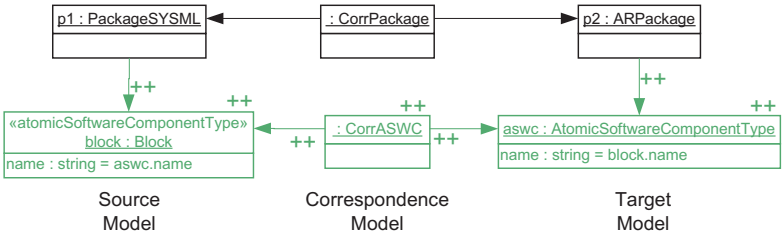


Figure 7: TGG rule for the transformation of a block to an atomic software component

Besides meta models also graph grammars can be used to describe a language. A graph grammar contains a set of graph grammar rules and a start graph, which defines the basic elements that must be contained in a model. The rules of a graph grammar consist of a *Left-Hand-Side* (LHS) and a *Right-Hand-Side* (RHS). The LHS defines the context, in which the rule can be applied, i.e. the elements that must already exist in the model. If the LHS of the rule can be matched to existing elements, these elements are replaced by the elements of the RHS. If the elements of the LHS are also part of the RHS, they are usually left untouched. Elements that occur only on the RHS are added to the model, elements that occur only on the LHS are deleted. Using a graph grammar, a model can be built by applying the rules successively, starting with the start graph.

Triple Graph Grammars combine three conventional graph grammars to describe the correspondence relationships between elements of two types of models. Two graph grammars describe the two models and a third grammar describes a correspondence model. Figure 7 shows a TGG rule for the transformation of a SysML block to an ASWC in AUTOSAR. This illustration also combines the LHS and RHS of the rule. The black elements belong to the LHS and the RHS of the rule. The elements marked with **++** (and printed green) belong only to the RHS and are created when the rule is applied. Rules that delete elements are not used in the context of model transformation with TGGs (cf. [Sch94]). The correspondence model is used to explicitly store correspondence

relationships between corresponding source and target elements. It allows to quickly find the target model elements corresponding to a given source model element.

Furthermore, TGGs allow bidirectional model transformations. The target model can be created from a source model (forward transformation) and vice versa (backward transformation). Besides model transformation, where a new target model is created, model synchronization is also supported. This means, that an existing target model is modified to make it consistent with a source model again. Modifications made to the target model are now retained and not overwritten[6]. Another advantage of synchronization is, that it can be performed much faster than a complete model transformation, especially if the models are large and only small modifications have to be synchronized.

The TGG rules are declarative and cannot be executed right away. Instead, operational transformation rules are derived for the forward and backward transformation. These operational rules are executed by a transformation engine to perform the model transformation. More information can be found in [GW09].

5 Architecture

In the industrial project, an architecture has been established, which integrates the tools TOPCASED and SystemDesk using the Eclipse platform. The tools are incorporated within the Eclipse platform in a way that both types of models (SysML and AUTOSAR models) exist in form of an EMF representation. Based on this EMF representation, the transformation and synchronization techniques described in Section 4 are realized. Subsequently, we describe this architecture in more detail.

5.1 Overall Architecture

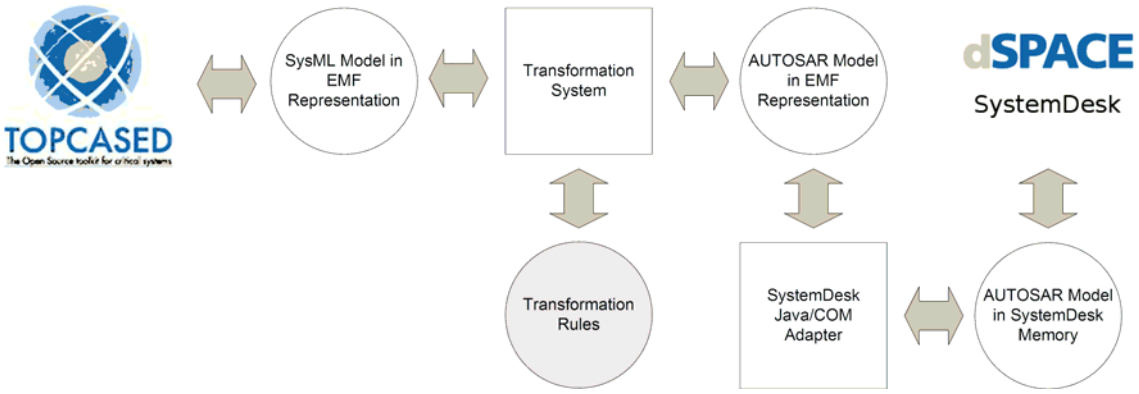


Figure 8: Overall system architecture

Figure 8 shows the overall architecture of the system. The core is the model transformation system, which implements the transformation and synchronization functionality. For this purpose, the transformation system needs to have access to the existing AUTOSAR and SysML models in EMF representation. In our architecture, this is possible in two different ways, in a file-based manner by reading and writing XML files, or alternatively by accessing the models directly in the modeling tools' memories. TOPCASED already uses EMF as its underlying modeling infrastructure and access to these EMF models from the transformation system can be realized without great effort. Accessing SystemDesk's models is more difficult because the technology gap between Eclipse/EMF and SystemDesk must be bridged. For this purpose, we developed a dedicated

[6] Unless they collide with changes in the source model. In this case, the changes of the target model are overwritten.

adapter, that reads an AUTOSAR model in EMF representation and writes it to SystemDesk, and vice versa. While the transformation system creates and modifies an AUTOSAR model (e.g., by a transformation or synchronization) in EMF representation, the SystemDesk Adapter takes care of reading and writing the model to and from SystemDesk. The model transformation and synchronization functionality is realized within the transformation system, which has access to the EMF models.

5.2 SystemDesk Adapter

SystemDesk provides an API, which is implemented in form of a Component Object Model (COM) and allows to access objects within SystemDesk from any COM-compatible application. Using this API also the AUTOSAR models within SystemDesk can be read and written. Based on the provided API, we have implemented an adapter, which is able to translate the model elements of the AUTOSAR Model in SystemDesk to EMF conformant model elements and vice versa. This adapter is used in our overall architecture to realize the bridge between SystemDesk and Eclipse/EMF like shown in Figure 8. In our current implementation, the adapter only partially supports the update and synchronization of AUTOSAR models in SystemDesk due to technical challenges. In the next version, we will implement the use of Unique Identifiers (UIDs), which are provided by the tool SystemDesk to fully support the update and synchronization of AUTOSAR models in SystemDesk.

5.3 Rules

The core transformation system of our architecture uses TGG rules like described in section 4 to realize the transformation and synchronization of SysML and AUTOSAR models. An example is the rule shown in Figure 7. This TGG rule transforms a SysML Block with the appropriate stereotype (*atomicSoftwareComponentType*) to an AUTOSAR ASWC. Such rules for the transformation and synchronization of the defined ports, interfaces and other constituents described in the meta model cutouts for SysML and AUTOSAR shown in Section 3 (and for the opposite direction) are also used within the transformation system.

6 Usage Scenarios

The described architecture supports several scenarios where, e.g., an initial AUTOSAR model is derived from an existing SysML model.

Additionally, the described architecture allows the synchronization of existing models by updating only changed model elements in the target model, without overwriting the whole model each time changes occur. Such a synchronization can be executed in both directions. Following, we describe different usage scenarios, in which the shown architecture allows an enhanced development process using model transformation and synchronization techniques.

6.1 Transformation from SysML to AUTOSAR

After the SysML model has been constructed, it needs to be transformed into an AUTOSAR model to get from the system design to an initial model for the software design. Design decisions concerning the software, which were defined in the SysML model have to be taken over to the AUTOSAR model. With the presented system such an initial AUTOSAR model can be automatically derived by a forward transformation. The automatic transformation is much faster than a manual transformation and there is less risk of introducing errors into the AUTOSAR model. A transformation in the other direction is also possible (backward transformation).

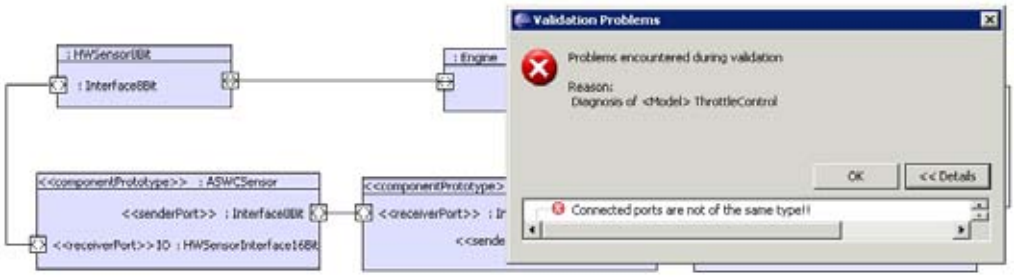


Figure 9: Screenshot of the OCL validation dialog in TOPCASED

6.2 Repeated Forward Synchronization from SysML to AUTOSAR

After the AUTOSAR model has been derived from the SysML model, modifications can still be made to the SysML model. These modifications have to be transferred to the AUTOSAR model, too. While the AUTOSAR model already exists, a complete retransformation is unnecessary. Therefore, only the modifications are synchronized. Furthermore, the AUTOSAR model might also have been modified, e.g., by changing the type of the IO port of the ASWC *ASWCSensor* shown in Figure 6. A complete retransformation would discard these modifications. Basically, our system supports such a synchronization. But due to restrictions of the current implementation of the adapter (compare Section 5) the models in SystemDesk are always overwritten. In the future, we will extend the adapter to allow the modification of the existing SystemDesk model.

6.3 Backward Synchronization from AUTOSAR to SysML

However, modifications may also be made to the AUTOSAR model in order to adjust the structure during refinement of the software architecture, e.g., to reuse an already existing component. Therefore, modifications also have to be propagated back to the SysML model. While most model transformation approaches are only permit unidirectional transformations, TGGs are bidirectional. So most changes are preserved in the SysML model.

How such a propagation of changes within the shown architecture using bidirectional transformation techniques supports the development process is demonstrated by the following scenario. When the type of the IO port of the ASWC *ASCWSensor* from Figure 6 is changed within the AUTOSAR model the TGG rules are triggered within the transformation system and the corresponding SysML IO port shown in Figure 4 is updated accordingly without overwriting the whole SysML model. When the SysML model is updated, the OCL constraint described in Section 3.1 is violated and an error message is automatically generated in TOPCASED (see Figure 9) that a SysML connector is connected to ports, which have a different type.

6.4 Iterative and Flexible Processes

The usage scenarios outlined in sections 6.1, 6.2 and 6.3 demonstrate that our approach can handle changes occurring in either model in any order. Therefore, the approach enables not only a strict sequential ordering, where first the SysML model is specified and thereafter the AUTOSAR model is derived from it (section 6.1). It also allows, that changes in the SysML model are propagated to the already existing AUTOSAR model (section 6.2) and that necessary changes in the AUTOSAR model are also accordingly adjusted in the SysML model (see 6.3). Therefore, instead of a rigid sequential process, also iterative and more flexible processes can be supported. Parallel development in the different phases is supported, changes in the different models can be synchronized and potential conflicts can be detected. Later changes of the AUTOSAR model will be reflected back to the SysML model after a synchronization. Such changes in an AUTOSAR model can lead to the violation of constraints existing in SysML model like described beforehand.

7 Conclusion & Future Work

SysML models employed early on and AUTOSAR models employed later in the process can be kept consistent using presented approach thanks to the use of model synchronization techniques. It has been outlined, that usage scenarios are feasible when employing our approach, and that additional flexibility concerning the process and in particular iterative development can be achieved.

As future work, we plan to further extend the coverage and also address other development artifacts. We also want to investigate how multiple models connected via model synchronization can be efficiently managed.

Acknowledgement

We thank the dSPACE GmbH for their support to develop the presented results and Oliver Niggemann, Joachim Stroop, Dirk Stichling and Petra Nawratil for their support in setting up and running the project.

References

[AUT07] AUTOSAR. *UML Profile for AUTOSAR*, January 2007. AUTOSAR GbR.

[BGN+04] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, August 2004.

[BHLW07] Simon M. Becker, Sebastian Herold, Sebastian Lohmann, and Bernhard Westfechtel. A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *Software and System Modeling*, 6(3):287–315, 2007.

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Daniel Varró. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In Julian Richardson, Wolfgang Emmerich, and Dave Wile, editors, *Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pages 267–270, Edinburgh, UK, 23 September 2002. IEEE Press.

[GW09] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8(1), 1 February 2009.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[JPB08] T. Johnson, C. Paredis, and R. Burkhart. Integrating Models and Simulations of Continuous Dynamics into SysML. 2008.

[OMG] OMG. *MOF QVT Final Adopted Specification, OMG Document ptc/05-11-01*. http://www.omg.org/.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of the* 20^th *International Workshop on Graph-Theoretic Concepts in Computer Science*, Herrschin, Germany, 1994. Spinger Verlag.

[Sys08] Systems Modeling Language v. 1.1, November 2008.

[Win07] Hans Windpassinger. Modellierungssprache für die Kfz-Software Entwicklung. Elektronik Praxis, 2007. http://www.elektronikpraxis.vogel.de/themen /embeddedsoftwareengineering/analyseentwurf/articles/95528/.

A textual domain specific language for AUTOSAR

Andreas. Graf (andreas.graf@itemis.de)
Markus Völter (voelter@acm.org)

Itemis
Blücherstr. 32
75177 Pforzheim
www.itemis.de

Abstract: AUTOSAR is a development partnership formed by leading OEMs and suppliers. The AUTOSAR specifications include a meta-model and a graphical notation that is used to specify E/E systems. However, experience from other software projects has shown that textual DSL have several advantages over graphical modelling. We address the problems of modelling AUTOSAR systems with graphical tools and show how these problems can be mended by a textual domain specific language. We suggest a number of steps that are necessary to design a textual DSL for AUTOSAR. This document focuses on AUTOSAR, but its results and conclusions can be applied to all complex embedded systems.

1 Background

To address the increasing complexity in automotive E/E systems, leading OEMs and suppliers have joined to form the AUTOSAR development partnership. The objective is to create an industry-wide standard that allows for collaboration on basic functions and infrastructure and still encourages innovation. AUTOSAR has published a number of specifications for E/E software systems, such as an Operating System (AUTOSAR Basic Software), a Runtime Environment for Applications (RTE) and an AUTOSAR meta-model that provides a formal specification for all developers for the description of AUTOSAR systems.

From the overall system-level down to the executable units (Runnables), the system is specified by means of models based on this standardized metamodel. An XML DTD is derived from the AUTOSAR metamodel to support model interchange, but there is no support for the interchange of diagram information. The current situation is very similar to the tooling of UML 1.x (which in practice never really worked that well). There is also no official specification of a model data store or an application framework for AUTOSAR authoring tools. Although there is the Artop[AR09] initiative, most tools still use proprietary databases or APIs. Customization to specific requirements would have to be redesigned and reimplemented for each available tool on the market.

2 Distributed Work

Several different roles are involved in the specification of the E/E system (or a partial system) of a vehicle. Most often, those roles are located at different sites. This separation of roles and teams introduces specific requirements for the interchange of models or model parts.

- **Function owners** are responsible for specifying the functions. One of their work products are interface and component descriptions in AUTOSAR format that are forwarded to the
- **SW-Developers.** They refine the specification with the internal component structure down to the executable units („Runnables“). During the iterations of development the interfaces might be changed by the functional owners. All models have to be kept in sync.
- **ECU and Basis-SW-suppliers** describe the features of the ECU and the interfaces of the basic SW by means of the AUTOSAR metamodel. The submodels are rather stable during the development, but have to be integrated into the model from a number of sources.
- **Integrators / Architects** specify network topology and the deployment of functions and software components to ECUs. The result is a complete model that can easily compass more than 100 ECUs, thousands of SW-components and ten thousands of signals.

3 Requirements on Modelling

Based on experience from similar sized projects that use classical metamodels like UML, we know that authoring tools often reach their limits with models of a comparable size. This is a consequence of shortcomings in tool implementations and scalable architectures, but also possibly a more conceptual issue. Classical development tools based on textual programming languages (such as the Eclipse) have scaled much better.

Hence, we propose to use textual domain specific languages for modelling AUTOSAR models. We like to define DSLs as follows:

A DSL is a focused, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are tailored to the stakeholders who specify that particular concern. A textual DSL is a DSL with a textual notation.

3.1 Model-based diff, merge and versioning

The problem of model comparison and merge has yet to be satisfactorily solved for graphical models. When the SW-developer receives an updated interface specification, he should easily be able to get information about the differences to his version and then merge those changes into his own model. This is not easily done for graphical models.

On the contrary, comparing and merging textual documents has a long history and is supported by a wide range of tools, specifically, those that are currently in use with many embedded development organizations. Consequently, version management is much better supported for textual documents than for graphical models.

3.2 Complexity

Complex systems such as entire vehicles or even the more complex subsystems cannot be displayed on one monitor in their entirety, much less can they be conveniently edited only by means of a graphical editor. A large number of connecting lines will cross the boundaries of the monitor. To avoid extensive scrolling, additional features for navigation have to be implemented: Hover texts could show the names of connected components or the details of port specifications. Also, components and their relationships are only one aspect of real-world models. A lot of information is more detailed and cannot easily be described graphically. In graphical environments this results in quite extensive property dialogs or name value pairs “in the box”. Both of these are not really a productive way of editing or rendering this information

Textual DSLs with tools akin to modern IDEs do much better here. However, there are situations where graphical notations are useful: In graph representations, the relationships between software components are very easily understood. Also, whenever you want to communicate to non-technical people, graphical languages are typically preferable because they are perceived to be “easier to understand”. However, there is a difference between graphical notation and graphical editing! Using tools like Graphviz, Prefuse or ZEST, you can easily render a textual model in a graphical way – without being able to edit in the graphical environment. This approach usually results in much better layout than manual tweaking and is easier to keep up-to-date. If the graph is shown directly in the IDE, a drill down facility can be provided easily (meaning that double-clicking on a node in the graph will bring you back to the location in the text file from which the respective node has been rendered).

3.3 Cost and Flexibility

Textual Editors and languages are much cheaper to design and implement then graphical editors (e.g. those built with Eclipse GMF). As a consequence, project specific customizations and new conceptual ideas are easier to implement.

3.4 Fast editing and import of large or many models

As an example, in one project, the limitations of graphical input became apparent. The software architects had to integrate existing non-AUTOSAR modules from several ECUs on a newer large ECU and at the same time migrate the software descriptions to the AUTOSAR format. Hundreds of software components and thousands of ports and signals had to be created. Graphical modelling requires a lot of switching between mouse and keyboard and accurate pointing on input fields, resulting in early fatigue. If part of a SW-system is already available (e.g. existing specifications or C-Code) and has to be brought into AUTOSAR form, it is advisable to use a textual editor.

3.5 Diagram interchange

Based on experience from UML models we see that the interchange of semantic models without the interchange of diagrams severely limits the readability and usability of the models. It works, as an interchange between the modelling tool and subsequence model processors (such as code generators) but it is not useful as an interchange between modelling tools and hence between development teams. At this time, AUTOSAR is in a situation very similar to UML 1.x. Models can be exchanged, but information on graphical layout is neither specified nor supported by tools.

4 Future Work

The use of textual DSL for specification of AUTOSAR models has a lot of benefits. The need has already arisen in AUTOSAR projects and project specific variants of DSLs have been implemented. The specification of a standardised concrete language involves a number of steps:

4.1 Identify stakeholders

As mentioned above, a DSL is tailored to the needs of stakeholders. As a first step, it is necessary to identify the stakeholders that would use a textual DSL. Architects and developers are more likely to use a textual representation compared to those roles that are involved in configuring the basic software. We expect less benefit of a DSL approach for basic software configuration than for the software architecture, since this part of the metamodel is less complex.

4.2 Define requirements for the language

Stakeholders are interviewed to collect requirements for the textual DSL. This critical step is essential for the acceptance of the new language and it will influence the workflow.

Comparison of our preliminary requirements research differs from other work like [HO07]: Höwing assumes that one of the use cases of an AUTOSAR DSL is the integrated specification of the software architecture and the program logic in one programming language. We expect the language for the static aspects to be separate from languages for program logic. The actual behaviour of software components is not only manually implemented in C, as addressed in the paper. Complimentary modelling tools like Matlab/Simulink or Ascet are in widespread use. An AUTOSAR solution needs features that support the integration of code from those models into the description of the system.

In addition, we expect a need for features that are currently not addressed in the released AUTOSAR specification. Specifically the support of variant management is expected to influence the design of a DSL. First implementations can be found e.g. in aXbench [AX09].

4.3 Define meta-model subset

From the set of stakeholders and the requirements, the subset of the AUTOSAR model that is to be described by a textual model is derived. It is the set of meta-model elements that are frequently used by the stakeholders.

4.4 Identify available tool features

The state-of-art of DSL tools influences the language constructs that could be use for the DSL. [PP08] contains a comparison of functional features of several modelling tools. For a full comparison according to the requirement, further comparison criteria like performance, licensing models, extensibility and others have to be introduced.

4.5 Identify technical basis in the industry

One major influence on the selection of a DSL tool is the tool environment that is already found in the industry. New tools should integrate into existing or emerging environments to avoid additional cost and effort. We intend to align our activities with the following two initiatives: Artop, which aims to provide a platform for AUTOSAR tooling, and the Eclipse Automotive Industry Working Group, which intends to provide an Eclipse distribution for Automotive tooling.

4.6 Define requirements for tools

Define the requirements that a DSL tool must meet to be able to process the AUTOSAR textual DSL. This might include features not yet widely available, if deemed necessary. Requirements of new features might be forwarded to the tool.

4.7 Define Language

The actual specification. The grammar for the DSL can obviously be handcrafted. To reduce effort and increase quality, we intend to assess the possibility of deriving the grammar from the AUTOSAR metamodel automatically.

4.8 Prototyping / Reference implementation

To make sure that the specified language meets all requirements, a prototype should be built as early as possible. Part of this step is the selection of an adequate tool.

5. Existing Experience

We would like to refer to two specific pieces of experience. One is the textual language used by people at BMW and BMW Car IT to define AUTOSAR models. As part of the pragma tool suite, a textual language, together with Ruby-based code generators has been implemented and used successfully in real world projects by real developers.

Second, one of the authors has used textual languages for describing software architectures in various domains successfully over the last year. From this work, a paper [VO08] has resulted, that shows the real world example, and also discusses the advantages and trade-offs of using textual languages for software architecture modelling.

References

[AR09] http://www.artop.org/

[AX09] http://axbench.isst.fraunhofer.de/

[HO07] F. Höwing: Effiziente Entwicklung von AUTOSAR-Komponenten mit domänenspezifischen Programmiersprachen. GI-Jahrestagung 2007: 551-556. Bremen, 2007.

[PP08] M. Pfeiffer, J. Pichler: A Comparison of Tool Support for Textual Domain-Specific Languages

[VO08] Markus Voelter: Architecture As Language, http://www.voelter.de/data/articles/ArchitectureAsLanguage-PDF.pdf

Demonstrating IEC 61508 Compliance in Model-Based Design (Work in Progress)

Ines Fey[1], Mirko Conrad[2]

[1] samoconsult,
Berlin, Germany
ines.fey@samoconsult.de

[2] The MathWorks, Inc.,
Natick, MA, USA
mirko.conrad@mathworks.com

Abstract: Automotive software components are increasingly engineered using Model-Based Design. OEMs and suppliers have begun to consider Model-Based Design to develop embedded software for applications that need to comply with IEC 61508-3.
For automotive software, IEC 61508-3 is often considered as state-of-the-art for high-integrity software within the industry. To demonstrate compliance with the standard, the objectives and recommendations outlined in IEC 61508-3 need to be mapped onto Model-Based Design approaches and tools.
To identify gaps w.r.t. standard compliance, the software engineering processes utilized for the software component under consideration are usually audited. Even for traditional software development processes this is not a straightforward activity. Due to the complexity and structure of the standard, gaps are usually not self-evident and hard to discover. IEC 61508-3 gap analysis in the context of Model-Based Design is even more challenging, since the standard was written with hand-coded software in mind. Appropriately prepared compliance documentation increases both the efficiency and effectiveness of IEC 61508 process audits.
This paper proposes a novel artifact centric, and template supported, compliance demonstration approach intended to ease IEC 61508-3 compliance documentation for software developed using Model-Based Design. Distinguishing between application specific and application agnostic compliance arguments facilitates partial re-use of the compliance documentation across projects using similar tool chains and processes, e.g. within product lines.

1 Model-Based Design for Automotive Software Components

Within the development of automotive software components, graphical modeling with Simulink® [SL] and Stateflow® [SF] can be used for the conceptual expectation of the functionality to be implemented. This way, the application part of the software to be developed can be modeled using time-based block diagrams (Simulink diagrams) and event-based state machines (Stateflow charts). Such a model of the application software can be simulated early within the software lifecycle. The **model** of the application software serves as the primary representation throughout multiple phases of a development process: Different elaborations of the model specify the desired functionality, provide design information and finally serve as the basis of the implementation by means of production code generation.

In practice, these different aspects are reflected in a step-wise transformation of the model of the application software from an early executable specification into a model suitable for production code generation and finally its automatic transformation into C code (**model evolution**) [CFG+05]. To accomplish this, the model of the application software must be enhanced by adding design information and implementation details. This way, application software development becomes the successive refinement of models followed by implementation through **automatic code generation**, compilation and linking as shown in Fig. 1.

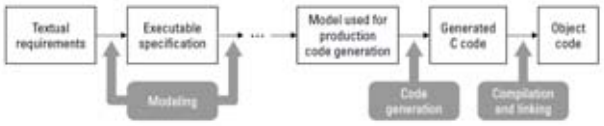


Fig. 1: Model-Based Design workflow

The modeling notations offered by Simulink and Stateflow are used throughout these consecutive modeling stages. Different verification and validation activities can be carried out at the model level; i.e., early in the software lifecycle before source code becomes available.

Note that this development paradigm exhibits certain specifics; e.g., the existence of new executable software engineering artifacts (i.e. models) and the partial blending of different lifecycle phases.

An application software component developed this way can be combined with other application software components to form the application software of the embedded system. In combination with the basic software and the embedded system's hardware the application software provides the functionality of the embedded system e.g. an automotive electronic control unit (ECU).

2 IEC 61508 and Model-Based Design

IEC 61508 'Functional safety of electrical / electronic / programmable electronic safety-related systems' was developed in the 1990s to define a generic, i.e. sector-independent, safety standard. By constraining the software engineering processes, part 3 of the standard [IEC 61508-3] attempts to reduce the number of faults introduced by the software engineering processes and to increase the number of faults revealed by the process. The degree of rigor required for software development and quality assurance depends on the criticality of the embedded software, or its safety integrity level (SIL). Depending on the SIL required, IEC 61508-3 recommends more than 100 specific techniques and measures for the different lifecycle phases. These techniques and measures are presented in 19 software safety integrity tables, where they are mapped to the four safety integrity levels, SIL 1 to SIL 4.

IEC 61508-3 dates back to 1998, when most software was hand-coded. As a result, it does not cover advanced software development technologies. When applied in the context of Model-Based Design [MBD, SL, SF] and code generation [RTW-EC], the standard must be mapped onto the specific development artifacts, processes, and tools used in Model-Based Design. IEC 61508 explicitly allows using an overall safety lifecycle that differs from the traditional software lifecycle the standard is based on as long as the objectives and requirements of each clause of the standard are met. However, the standard does not provide detailed guidance on how to map the requirements and objectives to onto new software development artifacts (namely executable models) and the partial blending of different development phases in Model-Based Design. This uncertainty frequently results in questions and compliance discussions.

To conform to IEC 61508-3, the applicant needs to demonstrate that the requirements have been satisfied to the required criteria specified (for example SIL) and therefore, for each clause or subclause, all the objectives have been met[1]. In practice, the software engineering processes utilized for the software component under consideration are audited to identify gaps. To facilitate this process, the applicant usually compiles **compliance documentation**.

Even for traditional software development processes this is not a straightforward activity. Due to the complexity and structure of the standard, gaps are usually not self-evident and hard to discover. IEC 61508-3 gap analysis in the context of Model-Based Design is even more challenging, since the standard was written with hand-coded software in mind. Efficiency and effectiveness of IEC 61508 process audits depend on appropriately prepared compliance documentation.

This paper proposes a novel **artifact centric, and template supported, compliance demonstration approach** intended to ease the IEC 61508-3 compliance documentation process for software developed using Model-Based Design. The approach is illustrated by using the 'model used for production code generation' as an example. The proposed artifact centric compliance demonstration is compared with the traditional approach using measure/technique centric software safety integrity tables.

3 Demonstrating IEC 61508-3 Compliance

IEC 61508 compliance can be certified by independent, external certification authorities, such as the TÜV. Some organizations 'self-certify' their products/ systems; i.e., require a demonstration of IEC 61508 compliance without external certification. Certain aspects of the standard may be relaxed or tightened in this case. In both cases, the applicant needs to **document compliance with the relevant set of IEC 61508 requirements**. For software, this traditionally involves creating tailored instances of the software safety integrity tables that explain how each recommended technique/measure was interpreted and applied for the software component under development.

[1] IEC 61508 compliance does not ensure the safety of the software or the system under consideration. However, IEC 61508 is considered as state-of-the-art or as Generally Accepted Rules of Technology for development of safety-related systems in various industries [Fal02, Lov06].

3.1 Using Tailored Software Safety Integrity Tables

The objectives and requirements outlined in IEC 61508-3 target the quality of the software engineering processes. Therefore, the standard constrains software engineering activities, the measures and techniques utilized, and the integration of specific safety activities.

Tailored software safety integrity tables are frequently used to document standard compliance based on the applied measures within the context of a particular project (Tab. 1). Annexes A and B of IEC 61508-3 provide 19 basic software safety integrity tables that list the techniques and measures recommended for each SIL (cf. columns #1 and #2 in Tab. 1). These tables are organized according to the different software lifecycle phases as outlined in the standard (**measure/technique centric structure**).

In order to demonstrate IEC 61508-3 compliance, these basic tables are extended by a new column to document how a given technique/measure was interpreted and applied for the application under consideration (cf. column #3 in Tab. 1). If a particular highly recommended technique or measure is not used, then a justification should be documented. The tailoring process for the software safety integrity tables is illustrated by means of worked examples in annex E of IEC 61508-6 [IEC 61508-6].

The resulting tailored tables as a whole are referred to as **compliance matrix** in engineering practice. They are used within the compliance demonstration process as evidence that the requirements and objectives of the standard have been met. The completed compliance matrix usually contains a mixture of application specific and application agnostic compliance arguments. Compliance matrices are often recreated from scratch for each application.

Technique/measure	SIL3	Interpretation in this application
(1a) Structured methods	++	Used to describe interaction between application software components and basic software
(1b) Semi-formal methods	++	Used for all application software components. In particular time-based block diagrams as provided by Simulink 7.3, and event-based state machines as provided by Stateflow 7.3
(1c) Formal methods	+	Only exceptionally, for application software components X and Y
(2) Computer-aided design tools	++	Used for the selected methods
(3) Defensive programming	++	All measures except those which are automatically inserted by the compiler are explicitly used in application software where they are effective
(4) Modular approach	++	Software module size limit, fully defined interface, …
(5) Design and coding standards	++	Use of MISRA-C:2004 coding standard, no dynamic objects, …

Tab. 1: Example of a tailored software safety integrity table
(following IEC61508-6, Table E.14)

Compliance matrices structured as in the given example, have proven themselves as helpful to support the compliance demonstration process for single, independent and self-contained applications. However, they are less suitable for automotive engineering organizations, which develop multiple, related applications following similar processes. The structuring of the tables is also less than ideal for Model-Based Design projects.

3.2 An Artifact Centric Approach for IEC 61508 Compliance Documentation

3.2.1 Artifact Centric Compliance Demonstration Tables

In Model-Based Design, core software development artifacts such as Simulink and Stateflow models evolve through multiple lifecycle phases (cf. section 1). In addition, more than one development team may work on the same artifact.

In order to provide optimal support for the development teams that are responsible for a particular artifact, all information relevant to assess the standard compliance of this artifact should be easily accessible through a single entry point for this artifact (in the sense of a one-stop shopping place). Such an **artifact centric structure** is not very well supported with the compliance demonstration means presented so far.

To support these engineering requirements the authors propose an **artifact centric approach for IEC 61508 compliance documentation**.

The idea is to first gather all major software engineering artifacts. As outlined in section 1, a typical Model-Based Design process may comprise the following core artifacts: Textual requirements, Model as executable specification, Model used for production code generation, Generated C Code, Object Code.

Second, a complete list of the associated IEC 61508-3 requirements is added for each core artifact. These associated requirements are derived from the main part of IEC 61508-3 as well as from the software safety integrity tables; i.e., from annexes A and B. Note that this is not necessarily a 1-to-1 mapping, some requirements may belong to more than one artifact. Others may not directly apply.

The first three columns of Tab. 2 illustrate the structure of an artifact centric compliance table by using the model used for production code generation as an example. To facilitate navigating, we divided the table into three sections, ‘Constructive Activities’ (i.e. software engineering activities), ‘Verification and Validation Activities’ (quality engineering and assurance activities) and ‘Supporting Activities’.

For each applicable section of IEC 61508-3 the requirements derived from the main part of the standard are listed first. After that, the recommended measures and techniques stated in the software safety integrity tables in annexes A and B (if applicable) are listed. The numbers preceding the applicable measures and techniques are derived from the annex A tables.

If an applicable measure/technique from a main table (that is from annex A) is further refined by a corresponding detailed table (that is from annex B) the detailed table is inlined into the main table to facilitate ease of use (for example, see entries for Tables A.4, B7, and B.9 in Tab. 2).

The standard itself does not clearly separate the requirements in the textual part from the recommended measures / techniques in the tables. Some aspects are covered both in the text and in the tables, others are only covered once. This results in certain redundancies within the table. However, based on their practical experience the authors consider this unpleasant, but unavoidable with the current edition of IEC 61508-3.

Third, the requirements, measures and techniques are mapped onto Model-Based Design processes and tools as they are used in this application. Some of the information in this column may be application independent; other parts may need to be augmented with evidence demonstrating the fulfilment of the corresponding requirements. If application specific information is required this should be clearly identified in the corresponding row of the table.

This is shown in the rightmost column of Tab. 2. To illustrate this aspect, Tab. 2 shows an example mapping for the constructive activities relevant to model development. Further information on the verification and validation activities for the model can be found in [CS09].

Applicant:	
Application / Program:	
Document Version:	
Date:	
Tool Versions Used:	Simulink® 7.3, Stateflow® 7.3, Real-Time Workshop® Embedded Coder™ 5.3, Simulink® Verification and Validation™ 2.5, Simulink® Design Verifier™ 1.4, PolySpace® Products for C 7.0, IEC Certification Kit 1.0 (R2009a/R2009a+)

Develop-ment artifact	**Technique/Measure, Associated Requirements**	**Ref. to IEC 61508-3**	**Model-Based Design processes and tools; Interpretation in this application, Evidence**
Model used for production code generation Model name: <application dependent> (see: <add link/ reference>	CONSTRUCTIVE ACTIVITIES		
	Software design and development – General Requirements	**7.4.2**	
	In accordance with the required SIL, the design method chosen shall possess features that facilitate: a) abstraction, modularity and other features which control complexity; b) the expression of: – functionality, – information flow between components – sequencing and time related information, – timing constraints, – concurrency, – data structures and their properties, – design assumptions and their dependencies; c) comprehension by developers and others who need to understand the design; d) verification and validation	7.4.2.2	a) - c) Provided by default for the selected tool chain: Simulink supports hierarchical decomposition and modularization of models on file level. Stateflow supports hierarchical decomposition. A differentiation between virtual (layout-only) and non-virtual architectural components needs to be provided. A list of model components (i.e. non-virtual subsystems, considered as modules) can be found at <application dependent>. Model Referencing facilitates a modular approach by including a model as a block in another model. Model referencing is used for the individual application software components. Model Reference Graphs are used to show the referenced models and their dependencies. d) See 'VERIFICATION AND VALIDATION ACTIVITIES' section
	The design method chosen shall possess features that facilitate software modification. Such	7.4.2.4	The Requirements Management Interface in Simulink Verification and Validation is used to support impact analysis.

Develop-ment artifact	Technique/Measure, Associated Requirements	Ref. to IEC 61508-3	Model-Based Design processes and tools; Interpretation in this application, Evidence
	features include modularity, information hiding and encapsulation.		Hierarchical modeling, and model referencing provide mechanisms for information hiding and encapsulation (see above).
	The design representations shall be based on a notation which is unambiguously defined or restricted to unambiguously defined features.	7.4.2.5	The design is represented within the model the by a limited number of notational elements such as subsystems and Stateflow charts. The graphical representation and the simulation feature support avoiding ambiguity.
	Where the software is to implement safety functions of different SILs, then all of the software shall be treated as belonging to the highest SIL, unless adequate independence between the safety functions of the different SILs can be shown in the design. The justification for independence shall be documented.	7.4.2.8	<application dependent>
	As far as practicable, the design shall include software functions to execute proof tests and all diagnostic tests in order to fulfil the safety integrity requirement of the E/E/PE safety-related system (as set out in IEC 61508-2).	7.4.2.9	<application dependent>
	The software design shall include, commensurate with the required safety integrity level, self-monitoring of control flow and data flow. On failure detection, appropriate actions shall be taken.	7.4.2.10	Simulink and/or Stateflow can be used to design fault detection checks on different levels in the value domain or in the time domain. <application dependent>
	Software design and development – Requirements for software architecture	**7.4.3**	
	…	...	…
	Software design and development - Requirements for support tools and programming languages	**7.4.4**	
	Technique(s)/Measure(s): (3) Language subset (4a) Certificated tools	Table A.3	(3) The MAAB V2.0 Style Guides (see: www.mathworks.com/industries/auto/maab.html) in conjunction with the department wide modeling guidelines (see: <add link/reference>) are used to define a subset of the modeling language. For critical components using Stateflow, the Stateflow language is restricted to Stateflow charts that implement either pure Mealy or Moore semantics. Model Advisor is used to partially enforce restricted language subsets for all model components. See list of Model Advisor checks

Develop-ment artifact	Technique/Measure, Associated Requirements	Ref. to IEC 61508-3	Model-Based Design processes and tools; Interpretation in this application, Evidence
			required for model component sign-off: <add link/reference>. See <add link/reference> for archived Model Advisor reports. Model reviews based on reports generated by Simulink Report Generator are conducted to check language subset considerations at the model level. See <add link/reference> for archived Model review reports. 4a) The utilized version of Real-Time Workshop Embedded Coder has been certified for use in IEC 61508 development processes.
	Software design and development – Requirements for detailed design and development, Requirements for code Implementation	**7.4.5, 7.4.6**	
	The software should be produced to achieve modularity, testability, and the capacity for safe modification.	7.4.5.3	Hierarchical modeling, model referencing, and subsystem masking provide mechanisms for information hiding and encapsulation.
	For each major component/ subsystem in the description of the software architecture design, further refinement of the design shall be based on a partitioning into software modules (i.e. the specification of the software system design). The design of each software module and the tests to be applied to each software module shall be specified.	7.4.5.4	A list of model components (i.e. non-virtual subsystems, considered as modules) can be found at <application dependent>. These model components are further refined using non-virtual subsystems.
	Appropriate software system integration tests should be specified to ensure that the software system satisfies the specified requirements for software safety at the required safety integrity level.	7.4.5.5	See software system integration test specification: <add link/reference>
	Technique(s)/Measure(s): (1b) Semi-formal methods • Finite state machines/state transition diagrams • Decision/truth tables (1c) Formal methods (2) Computer-aided design tools (4) Modular approach • Fully defined interface (5) Design standards	Table A.4, Table B.7, Table B.9	(1b) The model used for production code generation contains the detailed design. Each architectural element is hierarchically refined to the level of modules and basic operations. Simulink provides time-based block diagrams that can be used for detailed design. Stateflow provides extended finite state machine/ transition diagrams that can be used for detailed design. Stateflow provides Stateflow Classic Truth Tables and Embedded MATLAB® truth tables. Simulink provides a Combinatorial Logic block that implements a standard truth table.

Develop-ment artifact	Technique/Measure, Associated Requirements	Ref. to IEC 61508-3	Model-Based Design processes and tools; Interpretation in this application, Evidence
			(1c) Simulink Design Verifier – Property Proving can be used to formally verify detailed design considerations. (2) Simulink and Stateflow are used as computer-aided design tools for the detailed design. (4) Simulink supports hierarchical decomposition and modularization of models on file level. Stateflow supports hierarchical decomposition. A differentiation between virtual (layout-only) and non-virtual architectural components needs to be provided. A list of model components (i.e. non-virtual subsystems, considered as modules) can be found at <application dependent>. Model Referencing facilitates a modular approach by including a model as a block in another model. Model referencing is used for the individual application software components. Model Reference Graphs are used to show the referenced models and their dependencies. Model Advisor is used to verify that the top-level interface is fully defined. (5) …
	VERIFICATION AND VALIDATION ACTIVITIES		
	...	**...**	
	…	...	…
	SUPPORTING ACTIVITIES		
	...	**...**	
	….	...	…

Tab. 2: Example of an artifact centric compliance demonstration table

3.2.2 Using Templates to Facilitate Reuse

Organizations that run multiple software projects with similar development processes and tool chains look for compliance matrices that facilitate re-use and sharing of best practices (cf. [CD06,Con07]). Therefore it is desirable to factor out the application-independent parts and make them available for reuse by means of templates

As a first step to avoid starting from scratch each and every time when a new project starts, columns 1 … 3 could be provided as templates. They don't need to be changed as long as the list of artifacts remains unchanged.

Please note that different organizations may have different views on what the applicable requirements, measures and techniques for a given artifact are. Therefore the authors suggest that the applicant should get buy-in from all stakeholders (incl. certification authority) for this list.

In addition, column 4 contains a significant portion of application-independent information that can be reused across projects that share the same development processes and tool chains. This is appealing to larger development organizations and product lines. Under these circumstances, the central elements of the tool chain used in function development are usually fixed for a rather long period of time.

Under such circumstances it may be worth to replace or refine information in the 'Applicable Model-Based Design tools and processes' with information specific to the release of the Simulink product family [SL, SF] that is currently used for function development. That means to tailor and select techniques and tools to apply the standard.

Significant effort savings are expected if projects using those pre-filled templates as basis for their compliance demonstration process. Only project / application specific information such as references to actual work products and corresponding quality or status information need to be modified or included.

The proposed approach also results in 'standardized' compliance matrices. If the applicant that used such an approach works repeatedly with the same certification authority the assessors will get more familiar with the information provided over time. Efforts to clarify issues and miscommunications can be reduced greatly. The compliance assessment effort is being reduced.

Based on this information, each artifact that is in the scope of the project team can be assessed with respect to its associated requirements. Compliance with these requirements could be made part of the artifact's sign-off procedure.

4 Summary and Conclusion

At present, IEC 61508 is a widely used standard for high-integrity systems in the European automotive industry. The objectives and requirements of the standard apply to hand-coded software as well as to software developed using Model-Based Design. The fusion of software lifecycle phases in Model-Based Design and the major role of executable models within these phases require a specific interpretation of the standard if applied in the scope of Model-Based Design projects.

This interpretation affects the IEC 61508 compliance demonstration process as well. Carrying out the compliance documentation in the traditional way; i.e., by using and extending the measure/technique centric software safety integrity tables that come with IEC 61508-3, has various disadvantages.

The proposed artifact centric and template supported compliance demonstration approach to IEC 61508 compliance documentation in the context of Model-Based Design projects, provides an engineering solution to capture all requirements associated with a given software development artifact and to evaluate their realization in a given project. The proposed structure also facilitates a department or company wide deployment if the different projects standardize on the same processes, tool chains and versions. It also supports developers and quality managers in their day-to-day work.

It shall be noted, that the compliance demonstration discussed here is not sufficient to establish the overall safety of the system. It needs to be augmented by other documents and artifacts including a system-specific argumentation such as a safety case (see e.g. [RDG07]).

References

[Con07] M. Conrad: Using Simulink® and Real-Time Workshop® Embedded Coder for IEC 61508 Applications. White Paper, Safety Users Group, 2007 http://www.safetyusersgroup.com/documents/AR070002/EN/AR070002.pdf

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

[CFG+05] M. Conrad, I. Fey, M. Grochtmann, T. Klein: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. Inform. Forsch. Entwickl. 20(1-2): 3-10 (2005)

[CS09] M. Conrad, G. Sandmann: A Verification and Validation Workflow for IEC 61508 Applications. SAE Techn. Paper #2009-01-0271, SAE World Congress 2009

[Fal02] R. Faller: The Evolution of European Safety Standards. exida.com, 2002

[IEC 61508-3] IEC 61508-3:1998. Int. Standard Functional safety of electrical/ electronic/ programmable electronic safety-related systems - Part 3: Software requirements. 1998.

[IEC 61508-4] IEC 61508-6:1998. Int. Standard Functional safety of electrical/ electronic/ programmable electronic safety-related systems - Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3. 1998.

[Lov06] T. Lovric: Sicherheitsanforderungen an Fahrerassistenzsysteme. 2. Sachverständigentag (SVT 2006), Sept. 2006

[MBD] Model-Based Design web page. The MathWorks Inc., www.mathworks.com/applications/controldesign/description

[RGD07] W. Ridderhof, H.-G. Gross, H. Dörr: Establishing Evidence for Safety Cases in Automotive Systems – A Case Study. SafeComp 2007

[RTW-EC] Real-Time Workshop® Embedded Coder™ product page. The MathWorks Inc., www.mathworks.com/products/rtwembedded

[SF] Stateflow® Product Page. The MathWorks Inc., www.mathworks.com/products/stateflow

[SL] Simulink® Product Page. The MathWorks Inc., www.mathworks.com/products/simulink

An Executable and Extensible Formal Semantics for UML-RT

Stefan Leue and Wei Wei
Department of Computer and Information Science, University of Konstanz, D-78457 Konstanz, Germany
{Stefan.Leue, Wei.Wei}@uni-konstanz.de

Abstract: The semantics of a software modeling language is important. The syntax only defines a set of identifiers used to construct a model, as well as their placements and relations to each other. It is up to the semantics to interpret the meanings of those syntactic elements, e.g, how run-time entities are created, how they behave, and how they react to each other. A modeling language without semantics is the same as Java without compilers and JVM. Many industrially relevant software modeling languages lack a comprehensive, rigorous semantics. Even the semantics of some modeling aspects like schedulability are intentionally left open so that tool vendors and model designers can adopt different semantic variants for their own convenience. As a result, semantic ambiguities often lead to great difficulties in software development, validation, and code generation. Existing modeling language formalization approaches deal with semantic ambiguities in a naive way: they simply choose one particular semantic variant and ignore all other possibilities. The resulting formal semantics is therefore useful only for some CASE tools while useless for the others. To address this problem, we suggest a new formalization solution that handles semantic ambiguities in an explicit and extensible way so that different semantic variants can be easily adopted. This is achieved by making use of object-oriented concepts of inheritance and polymorphism. We illustrate our idea with a formal semantics that we gave for UML-RT, a dialect of UML for real-time systems. The given semantics is also executable and can be conveniently used in many kinds of software development practices like software validation and model-based testing.

1 Introduction

The use of software models has become more prevalent in today's software development. Ideally, a software model offers a simplified but *unambiguous* view of a software system. The usefulness of such a model can be manifold. First, software models serve as more precise documentation that facilitates a common and subtle understanding of the system among users, developers, testers, and designers. Second, software models can be used to validate software design and reveal potential design errors. Lastly, software models can be used to synthesize the final implementation code of the system, hereby reducing human errors injected during the coding phase.

However, all the above mentioned benefits cannot be enjoyed when software models have ambiguous meanings. First, an ambiguous model may result in inconsistent and incorrect understandings of the system. Second, formal verification and model-based testing

methods are only applicable to those modeling languages with rigorous semantics. Last, the ambiguities in a software model may lead to inconsistency between the model and the final implementation code. Consequently, the correctness of the software model may not necessarily be preserved in the final code.

Many industrially relevant software modeling languages, such as the Unified Modeling Language (UML) [OMG05], lack a comprehensive, rigorous semantics. Even the semantics of some modeling aspects like schedulability are intentionally left open, so tool vendors and model designers can adopt different semantic variations for their own convenience. There have been both academic and industrial efforts to give formal semantics for modeling languages, e.g., the UML 2 Semantics Project [Sem], among a great number of others. All these efforts had to struggle with the problem of semantic ambiguities of the modeling language being considered. Many formalization approaches adopted a naive solution by simply choosing one particular semantic variant, usually one as implemented in a particular CASE tool like IBM Rose RealTime (Rose RT), and ignoring all other possibilities [vdB06, RSM05, BH07]. Such a simple solution is problematic because the resulting semantics is only useful for some CASE tools while useless for others. Another problem is that most existing formal semantics are specified in a mathematical or logical framework which is hard to understand by average software developers [BH07, CMAT06]. Moreover, some semantics are not executable and their analysis methods cannot be fully automated [GBSS98, vdB06, RSM05]. This would further limit the usefulness of such semantics in real-life software development.

To illustrate our solution to the above problems, we designed an executable and extensible semantics for UML-RT, a dialect of UML for modeling real-time systems. The semantics is given in AsmL, a specification language developed at Microsoft research and supported by the Spec Explorer tool. The ambiguities of the UML-RT language is explicitly handled in our semantics using the object-oriented concepts of inheritance and polymorphism. Based on our semantics, we show how model validation can be now conveniently achieved. We will also discuss how to automate the demonstrated validation procedure, and suggest a model-based testing approach based on our semantics.

2 UML RT and the Existing Formal Semantics

UML-RT [SR98] was proposed as a UML dialect customized for the design of distributed embedded real-time systems [Sel99]. UML-RT is based on the ROOM notation [SGW94] and supported by the Rose-RT tool [Ros]. UML-RT finds applications in a broad range of systems [Her99, SFR97, GBC04, FNDR98]. Most of the modeling features of UML-RT have been incorporated into UML 2.

Figure 1 shows a simple UML-RT consisting of a set of concurrent autonomous actors. Actors exchange messages with one another through ports. A port is associated with a set of buffers at run time. It is however unspecified in UML-RT how ports are associated with buffers as well as how and when messages are stored, retrieved, and dispatched. These are examples of semantic variation points. The behavior of an actor is represented as a

Figure 1: A simple client-server model in UML-RT.

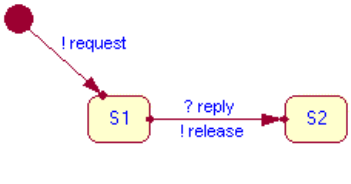


Figure 2: The state machine of the actor class `client`.

state machine. The firing of a transition is triggered by the reception of a message from a designated port. A transition may perform a certain action when it is taken.

There have been several attempts to assign formal semantics to UML-RT using various formalisms, such as [GBSS98, vdB06, KMR02, FOW01, EKHG01, RSM05, CMAT06, BH07]. All these existing semantics make the assumption that each port has one distinct buffer to store messages and message buffers are first-in-first-out (FIFO) queues. This may however lead to false conclusions during software validation. Consider our example in Figure 1. It is an important property for the model to satisfy that any client will eventually receive a reply from the server. Using the existing semantics, we will come to the conclusion that the property is satisfied, which seems very intuitive: The fact that any client will send a release message assures the server to be able to respond to the next client. Now the problem comes when we use the code generation feature of the Rose RT tool to obtain the implementation code from the model. When we execute the generated code, we will surprisingly find out that sometimes the property does not hold – one client will never get the reply from the server. This results from how Rose RT implements the message dispatching and scheduling mechanism. In Rose RT, every actor has only one FIFO buffer from which the actor retrieves messages. This buffer is shared among all the ports of the actor. When the head message in this buffer cannot be used to trigger any transition, it is simply removed from the buffer to make the next message available. Therefore, any request message that arrives at the server before the first release message will be lost when the server is waiting for the release message to come. Consequently, a client will never get its reply when its request has been removed.

The above example shows that the ambiguities in UML RT lead to different interpretations of the language, and that the disparities in different semantics may cause unexpected software validation result. Such risks can be totally avoided when we make it explicit which semantic variant is currently being used. This is one of the design goals of our UML-RT semantics that we explain in the next section.

3 A UML RT Semantics in AsmL

The ongoing SURTA project [LŞW08a] is aimed to give an executable semantics for UML-RT that can easily adopt various semantic variants, and to utilize the given semantics in automated real-life software development activities such as model validation and

model-based testing. Our choice of the formalization language is AsmL [Asm], an object-oriented software specification language. AsmL has sufficiently rich features to support the extensible architecture of our semantics while having a formal semantics specified for its core. Another motivation is the fact that AsmL is well supported by the model exploration tool Spec Explorer [Spe], which enables automated software validation and model-based testing.

At the heart of the proposed semantics lies an advanced architecture that makes use of the object-oriented concepts of inheritance and polymorphism to handle the ambiguities in UML-RT. An ambiguous modeling element, such as message buffers, is defined as either an interface or an abstract class as illustrate in Figure 3. Any concrete interpretation of the element can be realized as a concrete class implementing the interface or extending the abstract class. In our semantics, we realized several types of message buffers, such as FIFO buffers or multiset buffers (bags), that can be re-used to implement different message scheduling mechanisms. New types of message buffers can also be easily implemented.

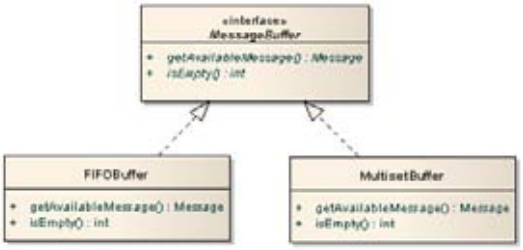


Figure 3: The semantics of message buffers in SURTA.

Another important characteristic of our semantics is to explicitly separate the syntactic representation of a modeling element from its semantics. There are three main advantages of the separation. First, the transformation of a UML-RT model into its syntactic representation is totally mechanic and can be fully automated. Second, since the semantic representation of a model is completely transparent to its syntactic representation, it is very easy to adopt different semantic variants. In fact there is no semantic representation for each individual model. The behavioral semantic of the UML-RT language is specified as a runtime environment that is used to execute all models guided by the respective syntactic representation. Lastly, it does not require a model to be re-transformed when a different semantic variant is used.

The above mentioned flexible architecture allows for far more modeling features of UML-RT to be formalized than other approaches. Significant examples are dynamic structures of actors, transition triggers and actions, replication, among others. We have implemented three semantic variants of UML-RT: (1) a variant representing the most general behavior of UML-RT models; (2) a variant based on Communicating Finite State Machines [BZ83]; and (3) a variant based on the Rose RT implementation. Details of the SURTA projects and the proposed semantics can be found in [LŞW08b].

4 Model Validation

With the support of Spec Explorer, we intend to use our UML-RT semantics to accomplish many software development activities in an automated way, such as model validation and model-based testing.

We first show how the simulation capabilities of Spec Explorer can be used to conveniently validate software models against certain properties. Consider the model in Figure 1 and the previously mentioned property that every client will eventually receive a reply from the server. Let's first see how this can be checked using the Rose RT tool. Rose RT can also be used to simulate models to reveal design errors. In order to check whether every client ever receives a reply message, we have to monitor the port of every client after each step of the execution of the model. This can be very tedious and time-consuming if there are a great number of client instances at runtime.

On the contrary, the above task can be easily achieved based on our semantics as follows. Given a desired property, we encode the negation of the property into a Büchi automaton. This automaton is then implemented as a special observer to automatically check after every single step of the model execution whether the property has been violated. We also implemented observers in a way that an erroneous trace of the model is reported when found.

The above mentioned validation approach based on our semantics can be fully automated. First, we are working on an automated translation algorithm to transform a UML-RT model into its syntactic representation in AsmL. Second, when a property can be expressed in an LTL formula, there are well-known algorithms to transform the negated property into a Büchi automaton, e.g., [SB00]. The challenge here is to generate succinct, efficient, and highly reusable validation code from those resulting automata.

5 Model-Based Testing

Rose-RT provides virtually no direct support for test case generations from Rose-RT models. We suggest how our UML-RT semantics can be used in model-based testing with the Spec Explorer tool [CGN+05].

In model-based testing, it is important to establish the mapping between a software model and the implementation under test (IUT): the mapping between state variables in the model and variables in the IUT; the mapping between actions in the model and the functions/procedures in the IUT. This relationship can be sometimes not so obvious and therefore hard to obtain.

The above mapping problem can be easily solved in the model-based testing approach based on our UML RT semantics: Given a Rose RT model, we generate its syntactic representation in AsmL using an automated transformation algorithm, and synthesize the IUT by the code generation feature of the Rose RT tool. Because the code synthesized by Rose RT preserves the state/transition structure of the original model, the mapping between the

IUT and the AsmL representation is quite straightforward. Such a direct mapping can be greatly utilized in automated test harness generation and test result interpretation. We can use the Spec Explorer tool to explore the state space of the AsmL representation of the model and select a certain set of traces as test cases. It is in our future work to develop trace selection algorithms with respect to different test coverage criteria. The selected test cases are then used to generate the test harness that guides the executions of IUT. The result of each IUT execution is then interpreted back in the context of the AsmL representation of the model in order to find any inconsistency with the model execution.

6 Conclusion

In this paper we argue the necessity to treat semantic variants more explicitly. We suggest an extensible formal semantic specification framework based on the object-oriented concepts of inheritance and polymorphism, in which different semantic variants can be easily implemented and adopted. The advantage of the proposed framework is then demonstrated by an executable semantics that we gave for UML-RT. We also discuss how our UML-RT semantics may be used in automated software development practices such as model validation and model-based testing.

References

[Asm] AsmL – Abstract State Machine Language (Microsoft). `http://research.microsoft.com/fse/asml`.

[BH07] J. Bezerra and C. M. Hirata. A Semantics for UML-RT using π-calculus. In *Proc. RSP 2007*, pages 75–82, 2007.

[BZ83] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, 1983.

[CGN+05] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-based testing of object-oriented reactive systems with Spec Explorer. Technical Report MST-TR-2005-59, Microsoft Research, 2005.

[CMAT06] M. I. Capel, L. E. M. Morales, K. B. Akhlaki, and J. A. H. Terriza. A Semantic Formalization of UML-RT Models with CSP+T Processes Applicable to Real-time Systems Verification. In *Proc. JISBD*, pages 283–292, 2006.

[EKHG01] G. Engels, J. M. Küster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *ESEC / SIGSOFT FSE*, pages 186–195. ACM Press, 2001.

[FNDR98] M. Fuchs, D. Nazareth, D. Daniel, and B. Rumpe. BMW-ROOM: An object-oriented method for ASCET. In *SAE'98*. Society of Automotive Engineers, 1998.

[FOW01] C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In *FASE 2001*, volume 2029 of *LNCS*. Springer Verlag, 2001.

[GBC04] Q. Gao, L.J. Brown, and L.F. Capretz. Extending UML-RT for control system modeling. *American Journal of Applied Sciences*, 1(4):338–347, 2004.

[GBSS98] R. Grosu, M. Broy, B. Selic, and G. Stefanescu. Towards a calculus for UML-RT specifications. In *Proc. OOPSLA*, 1998.

[Her99] D. Herzberg. UML-RT as a candidate for modeling embedded real-time systems in the telecommunication domain. In *UML'99*, volume 1723 of *LNCS*, pages 330–338. Springer, 1999.

[KMR02] A. Knapp, S. Merz, and C. Rauh. Model checking timed UML state machines and collaborations. In *FTRTFT'02*, volume 2469 of *LNCS*, pages 395–416. Springer, 2002.

[LŞW08a] S. Leue, A. Ştefănescu, and W. Wei. An AsmL Semantics for Dynamic Structures and Run Time Schedulability in UML-RT. In *Proc. TOOLS (46)*, volume 11 of *LNBIP*, pages 238–257. Springer, 2008.

[LŞW08b] S. Leue, A. Ştefănescu, and W. Wei. An AsmL Semantics for Dynamic Structures and Run Time Schedulability in UML-RT. Technical Report soft-08-02, University of Konstanz, 2008. Available from `http://www.inf.uni-konstanz.de/soft/publications_en.php`.

[OMG05] OMG. UML 2.0 superstructure specification. Technical Report formal/05–07–04, Object Management Group, 2005. Online at: `http://www.uml.org`.

[Ros] Rational Rose RealTime tool. Shipped within Rational Rose Technical Developer: `http://www.ibm.com/software/awdtools/developer/technical`.

[RSM05] R. Ramos, A. Sampaio, and A. Mota. A semantics for UML-RT active classes via mapping into Circus. In *FMOODS'05*, volume 3535 of *LNCS*, pages 99–114. Springer, 2005.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *Proc. CAV 00*, volume 1855 of *LNCS*, pages 248–263. Springer-Verlag, 2000.

[Sel99] B. Selic. Turning clockwise: using UML in the real-time domain. *Comm. of the ACM*, 42(10):46–54, Oct. 1999.

[Sem] The UML 2.0 Semantics Project (1.1.2005–31.12.2006). `http://www.cs.queensu.ca/~stl/internal/uml2/`.

[SFR97] M. Saksena, P. Freedman, and P. Rodzewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 240–25. IEEE Computer Society, 1997.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[Spe] Spec Explorer tool. `http://research.microsoft.com/SpecExplorer`.

[SR98] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. `http://www.ibm.com/developerworks/rational/library/139.html`, March 1998.

[vdB06] M. von der Beeck. A Formal Semantics of UML-RT. In *Proc. MoDELS*, volume 4199 of *LNCS*, pages 768–782. Springer, 2006.

TUDOOR - Ein Java Adapter für Telelogic DOORS®

Jae-Won Choi, Anna Trögel, Ingo Stürmer

Model Engineering Solutions GmbH

Abstract: Im Bereich des Requirements Engineering hat sich DOORS® der Firma Telelogic als Marktführer durchgesetzt. Externe Applikationen, die auf Funktionalitäten zurückgreifen wollen, die DOORS nicht zur Verfügung stellt, wie z.B. Verlinkung der Anforderungen zu Test- und Modellierungswerkzeuge von Software oder die Verwaltung von DOORS Modulen, mussten bisher die DOORS-spezifischen DXL-Scriptsprache oder die C-API von DOORS verwenden. Für Java-Anwendungen wird derzeit keine direkte Unterstützung von DOORS geboten. Daher haben wir TUDOORS entwickelt, eine allgemeine Java Schnittstelle für DOORS, die auf einem DOORS Meta-Modell basiert. Der Beitrag stellt TUDOOR in seiner Gundfunktionalität vor, beschreibt dessen Anwendung, sowie Auszüge aus dem DOORS Meta-Modell und den Prozess der Java Adapter-Generierung.

1 Der Java Adapter TUDOOR

Das Werkzeug DOORS® der Firma IBM/Telelogic ist de-facto-Marktführer im Bereich des Requirements Engineering für Software-Projekte. DOORS wird z.B. zur Definition und Strukturierung von Anforderungen an Software(-Systeme) verwendet. Die darüber hinaus benötigten Funktionalitäten, wie z.B. Dokumentengenerierung, Verlinkung der Anforderungen zu Test- und Modellierungswerkzeuge von Software, kurz: die Anbindung externer Applikationen an DOORS, wird für Java Applikationen nicht direkt unterstützt. DOORS bietet eine C-API und eine Scriptsprache, die *DOORS eXtension Language* (DXL) an, die den externen, lesenden und schreibenden Zugriff auf DOORS erlaubt. DXL ist im Vergleich zu gängigen Programmiersprachen, wie Java oder C, eine wenig verbreitete Scriptsprache, die nur von Experten verstanden wird. Die C-API stellt eine sinnvolle Alternative zu DXL dar, ist aber nicht mehr zeitgemäß, da heutzutage Java der de-facto-Standard ist, um Anwendungen zu entwickeln. Aus diesem Grund haben wir *TUDOOR* [MES09] entwickelt, eine generische Java Schnittstelle, die den lesenden und schreibenden Zugriff auf DOORS über Java-Klassen ermöglicht.

Der Zugriff auf DOORS über Java-Klassen bietet folgende Vorteile: (1) Man benötigt nur noch Java-Kenntnisse, um mit externen Anwendungen an DOORS anzukoppeln. Dies erlaubt die effiziente und kostengünstige Entwicklung von Add-Ons z.B. zur Analyse und Visualisierung von Anforderungsdokumenten; (2) Die Produktivität der Entwickler steigert sich deutlich, da die Kopplung an DOORS nicht mehr umständlich in einer Skriptsprache implementiert werden muss. Ferner ist eine Ankopplung an Standard-Java-Bibliotheken (z.B. für GUI Visualisierungen) wesentlich einfacher; (3) Die Schnittstelle unterstützt den direkten Zugriff auf Meta-Informationen sowie die Formulierung generischer Algorithmen, die mit Script-Sprachen kaum oder nur sehr schwer realisiert werden können.

Das Grundprinzip von TUDOOR bzw. des Java Adapters ist in Abb. 1 gezeigt. Der Adapter besteht aus Java- und DXL-Dateien, die in einem JAR-File bereitgestellt werden. Die Java Dateien enthalten alle Klassen, die zur Anbindung an DOORS benötigt werden. Diese Java-Klassen basieren auf dem DOORS Meta-Modell (siehe Kapitel 2), das alle DOORS Objekte, Daten und Methoden beschreibt. Das JAR-File selbst wird von der externen Java-Applikation verwendet, die auf DOORS zugreifen möchte.

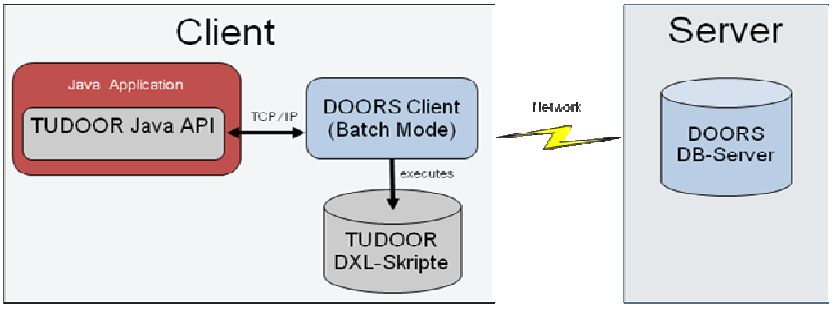


Abb.1 – Funktionsweise und Kommunikationswege von TUDOOR

Die Java-Klassen des Adapters kapseln die DXL Befehle, die für den direkten Zugriff auf DOORS benötigt werden. Da alle verfügbaren DXL Befehle einen zu großen Funktionsumfang für den einzelnen Nutzer ermöglichen, wie z.B. gewolltes bzw. ungewolltes Auslesen oder Löschen von Inhalten, für die keine Berechtigung besteht, stellt der Adapter eine nur lesbare DXL Whitelist bereit, die alle erlaubten DXL Befehle enthält. D.h. der Funktionsumfang ist auf ein sicheres und erlaubtes Subset der DXL Scriptsprache eingeschränkt.

2 DOORS Meta-Modell

Dieses Kapitel stellt das in TUDOOR zu Grunde liegende DOORS Meta-Modell vor. Um die Integration und den Austausch von Daten verschiedener Applikationen zu ermöglichen, ist die Verwendung von Standards zum Verwalten von Metadaten praktikabel. Eine einheitliche Repräsentation von Metainformationen sollte dafür sowohl mehrere Abstraktionsebenen von Metadaten, als auch generische, präzise definierte Sprachelemente abbilden können. Eine solche Sprache bietet die von der OMG entworfene Meta Object Facility (MOF) [OMG06], die mit der graphischen Notation von UML erstellt wird. Das DOORS Meta-Modell basiert auf einer formalen Beschreibung von DOORS in Form eines MOF-konformen UML Modells (vgl. Abb. 2). Auf diese Weise kann die Struktur des Werkzeugs auf einer abstrakten Meta-Ebene plattform- und sprachunabhängig beschrieben werden, auch besser bekannt als *Platform Independent Model* (PIM) aus dem MDA-Bereich. Mit Hilfe des in der MOF-Spezifikation definierten XMI[1]-Formats kann dieses Meta-Modell aus dem jeweiligen Modellierungswerkzeug exportiert und in beliebige andere Formate übertragen werden. Daraus ergibt sich der Vorteil mit dem

[1] XMI – XML Metadata Interchange (OMG)

einmalig erstellten DOORS Meta-Modell Adapter für Anwendungen verschiedener Programmiersprachen entwickeln zu können. Für den in Java implementierten DOORS-Adapter werden mit der vom Java Community Process (JCP) verabschiedeten Spezifikation JMI[2]-konforme Java Schnittstellen generiert. Deren Implementierung und die Instanziierung der Klassen erlaubt das Erzeugen, Verwalten, Ändern und Löschen der Daten in DOORS. Beschrieben wird der Workflow von der Meta-Modell Erstellung bis hin zur Schnittstellengenerierung und Implementierung in Kapitel 3.

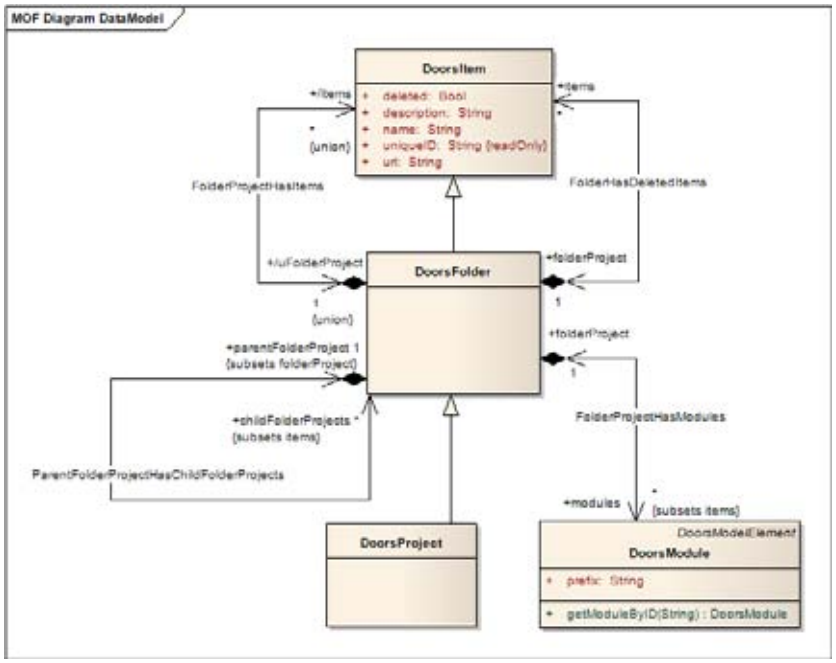


Abb. 2: Ausschnitt des DOORS Meta-Modells (Datenmodell)

Das in Enterprise Architect modellierte Meta-Modell gliedert sich in mehrere Pakete, die verschiedene Funktionalitäten von DOORS widerspiegeln, sowie DOORS Datentypen definieren: (1) DataModel, (2) DisplayControl, (3) BaselineControl, (4) AccessRights, (5) BaseTypes.

Im Datenmodell sind alle wesentlichen Elemente der DOORS-Struktur möglichst exakt[3] abgebildet. UML Klassen repräsentieren darin DOORS Elemente, Attribute ihre jeweiligen Eigenschaften und Assoziationen ihre Beziehung zueinander. Abb. 2 zeigt beispielhaft einen Ausschnitt des Datenmodells. Die Klasse *DoorsFolder* erbt demzufolge die Eigenschaften, wie Name (*name*) und Beschreibung (*description*), eines *DoorsItems* und führt wiederum eine Liste mit Kindelementen vom Typ *DoorsItem*, wodurch die hierarchische Ordnerstruktur von DOORS abgebildet wird. Jeder *DoorsFolder* hat zusätzlich eine Liste von *DoorsModules*. Ein *DoorsModule* muss von *DoorsItem* erben. Sowohl

[2] JMI – Java Metadata Interface (SUN)
[3] Im Zweifelsfall erfolgt hier eine Klärung in Zusammenarbeit mit dem DOORS Support.

DoorsFolder als auch *DoorsProject* haben eine Liste von Modulen. Diese Beziehung wird durch die Assoziation *FolderProjectHasItems* abgebildet. Das Paket *Display-Control* bildet die Funktionalität der Views[4] ab, die für Formal und Descriptive Module erstellt werden können. Im *BaselineControl* Paket ist die Möglichkeit Zwischenstände von Modulen zu speichern und im Access Paket die Zugriffsberechtigungen auf DOORS Elemente modelliert. DOORS Objekttypen sind im Paket BaseTypes definiert.

3 Generierung des Java-Adapters

Abb. 3: Workflow: Generierung des TUDOOR Java Adapters

MOFLON [AKRS08] ist ein in Java entwickeltes Meta-CASE Tool, das im Rahmen des gleichnamigen Forschungsprojekts an der TU Darmstadt, Fachgebiet Echtzeitsysteme, entstanden ist. Das Tool wird derzeit aktiv weiterentwickelt und dient zur Modellierung von MOF-konformen Modellen und deren Transformationen. Es bietet darüber hinaus die Möglichkeit MOF-konforme Modelle, die in anderen Applikationen modelliert wurden, über das XMI-Format zu importieren und durch Transformationen Java-Code zu generieren. Im Falle Java Adapters für DOORS wird mit Hilfe von MOFLON aus dem zuvor in Enterprise Architect erstellten Meta-Modell Java-Code generiert.

Der Workflow für die Generierung des Adapters ist in Abb. 3 gezeigt. Zunächst werden die benötigten Funktionalitäten im DOORS EA Meta-Modell hinterlegt. Hierbei wird versucht das Verhalten von DOORS möglichst exakt abzubilden[5]. Über das MOFLON Framework werden JMI-konforme Schnittstellen für die Java-Klassen generiert. Anschließend werden Funktionalitäten der Java-Klassen manuell implementiert.

---

[4] Views definieren im DOORS Client unterschiedliche Sichten auf Module. So können in Views unter anderem angegeben werden, welche Attribute eines Moduls sichtbar sein sollen.
[5] Im Zweifelsfall erfolgt hier eine Klärung in Zusammenarbeit mit dem DOORS Support.

4 Zusammenfassung und Ausblick

Die Adaptergenerieurung aus einem Meta-Modell für die Kopplung von Werkzeugen wird zukünftig eine wichtigere Rolle einnehmen. Die Entwicklung von TUDOOR zeigt, dass die modell-getriebene Softwareentwicklung einen Teil der Entwicklungsarbeit abnehmen und dadurch bestimmte Prozesse vereinfachen und weniger fehleranfällig gestalten kann. Durch die Generierung von Schnittstellen und Standardimplementierungen entfallen Routinearbeiten und Flüchtigkeitsfehler in diesen Teilen der Architektur. Dies setzt jedoch ein zuvor korrekt erstelltes Meta-Modell voraus.

Der Adapter lässt viel Spielraum für zukünftige Erweiterungen, wie z.B.: (1) Transaktionskonzept: Derzeit werden über die setter-Methoden Daten sofort in die DOORS Datenbank geschrieben. Sinnvoll wäre hier ein Transaktionsmechanismus, bei dem der Beginn und das Ende einer solchen Transaktion explizit angegeben werden kann. So wären Rückfalloptionen möglich, falls während einer Transaktion etwas nicht wie gewünscht verläuft oder diese explizit abgebrochen wird; (2) Verschlüsselte Kommunikation**:** DOORS wird auch für die Verarbeitung sensibler Daten eingesetzt. Daher stellt die verschlüsselte Übertragung der Daten eine wertvolle Ergänzung dar. Hiermit kann sichergestellt werden, dass die Kommunikationswege vor unerlaubten Zugriffen abgesichert werden; (3) Bidirektionale Kommunikation**:** Eine weitere sinnvolle Ergänzung ist die bidirektionale Kommunikation zwischen dem DOORS Client und dem Adapter. Sie soll es ermöglichen, Veränderungen die über den DOORS Client gemacht werden, direkt an den Adapter zu senden. So wäre sichergestellt, dass die im Adapter geladenen Daten immer aktuell sind, ohne aufwendige Polling-Mechanismen in periodischen Abständen durchführen zu müssen.

Literatur

[AKRS08] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, A. Schürr: "Metamodel-based Tool Integration with MOFLON", 30th International Conference on Software Engineering, New York: ACM Press, 05 2008, ACM Press, 807-810.

[Daimler08] Daimler AG, Evaluierung vorhandener Java-DOORS-Schnittstellen, interner Bericht, Juni 2008.

[MES09] Model Engineering Solutions GmbH, TUDOOR, Produktinformation, http://www.model-engineers.com/our-products.html, 2009.

[OMG06] Object Management Group. Meta Object Facility (MOF), Core Specification, January 2006.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2005-07

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-Workshops 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 (Hrsg.)	Tagungsband Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme III
2007-02	J. Rang	Design of DIRK schemes for solving the Navier-Stokes-equations
2007-03	B. Bügling, M. Krosche	Coupling the CTL and MATLAB
2007-04	C. Knieke, M. Huhn	Executable Requirements Specification: An Extension for UML 2 Activity Diagrams
2008-01	T. Klein, B. Rumpe (Hrsg.)	Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, Tagungsband
2008-02	H. Giese, M. Huhn, U. Nickel, B. Schätz (Hrsg.)	Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV
2008-03	R. van Glabbeek, U. Goltz, J.-W. Schicke	Symmetric and Asymmetric Asynchronous Interaction
2008-04	M. V. Cengarle, H. Grönniger B. Rumpe	System Model Semantics of Statecharts
2008-05	M. V. Cengarle, H. Grönniger B. Rumpe	System Model Semantics of Class Diagrams
2008-06	M. Broy, M. V. Cengarle, H. Grönniger B. Rumpe	Modular Description of a Comprehensive Semantics Model for the UML (Version 2.0)
2008-07	C. Basarke, C. Berger, K. Berger, K. Cornelsen, M. Doering J. Effertz, T. Form, T. Gülke, F. Graefe, P. Hecker, K. Homeier F. Klose, C. Lipski, M. Magnor, J. Morgenroth, T. Nothdurft, S. Ohl, F. Rauskolb, B. Rumpe, W. Schumacher, J. Wille, L. Wolf	2007 DARPA Urban Challenge Team CarOLO - Technical Paper
2008-08	B. Rosic	A Review of the Computational Stochastic Elastoplasticity
2008-09	B. N. Khoromskij, A. Litvinenko, H. G. Matthies	Application of Hierarchical Matrices for Computing the Karhunen-Loeve Expansion
2008-10	R. van Glabbeek, U. Goltz, J.-W. Schicke	On Synchronous and Asynchronous Interaction in Distributed Systems
2009-01	H. Giese, M. Huhn, U. Nickel, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme V