Budapest University of Technology and Economics
Department of Automation and Applied Informatics

PROCEEDINGS OF THE WORKSHOP ON MULTI-PARADIGM MODELING: CONCEPTS AND TOOLS 2006 (MPM06)

Editors:
Holger Giese
Tihamér Levendovszky

BME-DAAI Technical Report Series Volume 2006/1

Holger Giese
Tihamér Levendovszky (Eds.)

Proceedings of the Workshop on Multi-Paradigm Modeling: Concepts and Tools 2006 (MPM06)

Genova, Italy, October 3, 2006.

Proceedings of the Workshop on Multi-Paradigm Modeling: Concepts and Tools 2006 (MPM06)

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

1111 Budapest, Goldmann Gy. tér 3.

url	http://avalon.aut.bme.hu/mpm06
e-mail:	tihamer@aut.bme.hu
tel:	+36(1)4632870
fax:	+36(1)4632871

Editors:
Holger Giese
Tihamér Levendovszky

ISBN 963 420 878 9

Program Committee

Michael von der Beeck
BMW (DE)

Jean Bezivin
Université de Nantes (FR)

Heiko Dörr
DaimlerChrysler AG (DE)

Jean-Marie Favre
Institut d'Informatique et Mathématiques Appliquées de Grenoble (FR)

Reiko Heckel
University of Leicester (UK)

Jozef Hooman
University of Nijmwegen (NL)

Gabor Karsai
Vanderbilt University (US)

Anneke Kleppe
University of Twente (NL)

Ingolf H. Krüger
University of California, San Diego (US)

Thomas Kühne
Technical University Darmstadt (DE)

Juan de Lara
Universidad Autónoma de Madrid (ES)

Jie Liu
Microsoft Research (US)

Mark Minas
University of the Federal Armed Forces (DE)

Oliver Niggemann
dSPACE GmbH (DE)

Pieter Mosterman
The MathWorks (US)

Bernhard Schätz
TU Munich (DE)

Andy Schürr
Technical University Darmstadt (DE)

Hans Vangheluwe
McGill University (CA)

Bernhard Westfechtel
University of Bayreuth (DE)

Table of Contents

Invited Talk

Regular Papers

Preface

Today complex software-based systems often integrate different, previously isolated subsystems where different aspects such as the dynamic behavior or static structure are captured by notations using different formalisms (e.g. statecharts and user interface models, block diagrams for control, ...) at different level of abstractions. Therefore, multiple modeling paradigms have to be integrated for their model-driven development. This is especially true when - besides general purpose languages such as UML - domain specific languages are also employed.

Multi-Paradigm Modeling (MPM) aims to simplify the modeling of complex systems by combining three different directions of research:

- *Meta-Modeling*, which is the process of modeling formalisms.
- *Model Abstraction*, concerned with the relationship between models at different levels of abstraction.
- *Multi-Formalism Modeling*, concerned with the coupling of and transformation between models described in different formalisms.

This very first workshop on multi-paradigm modeling addresses the research related to the issues above as well as the clarification of the basic notions of multi-paradigm modeling.

We would like to thank all of the people who submitted papers as well as the Program Committee members who reviewed the submissions.

We wish you a pleasant workshop and nice discussions!

Holger Giese
Tihamér Levendovszky
Chairs

Invited Talk

Foundations and Challenges of Multi-Paradigm Modelling

Hans Vangheluwe
McGill University,
Montréal, Québec, Canada

Abstract. Engineering and Science invariably use models to describe structure as well as behaviour of systems. Models may have components described in different formalisms, and may span different levels of abstraction. In addition, model transformation is commonly used to transform models into domains/formalisms where certain questions can be easily answered. These various aspects are condensed into the term "multi-paradigm modelling".

Multi-paradigm concepts are easily applicable to "domain-specific modelling".

Using domain-specific modelling environments maximally constrains users, allowing them, by construction, to only build syntactically correct models.

Furthermore, the domain-specific, often visual syntax used matches the users' mental model of the problem domain. The time required to construct domain/formalism-specific modelling and simulation environments can however be prohibitive. Thus, rather than using domain-specific environments, users resort to generic environments.

Such generic environments are necessarily a compromise.

In this presentation, the foundations of (domain-specific) multi-paradigm modelling will be presented. It will be shown how all aspects of modelling can be explicitly (meta-)modeled enabling the efficient synthesis of domain-specific multi-paradigm modelling environments. Various scientific challenges and open problems such as the management of model evolution, and the modularisation of transformation models will be presented. In the examples, our Computer Automated Multi-Paradigm Modelling (CAMPaM) tool $AToM^3$ (A Tool for Multi-formalism and Meta Modelling) will be used.

Regular Papers

A Multi-Paradigm Modeling Approach for Reconfigurable Mechatronic Systems

Stefan Henkler
Department of Computer Science
University of Paderborn
Paderborn, Germany
shenkler@uni-paderborn.de

Martin Hirsch
Department of Computer Science
University of Paderborn
Paderborn, Germany
mahirsch@uni-paderborn.de

Abstract. Involved disciplines during the development of reconfigurable mechatronic systems are control engineering, electrical engineering, mechanical engineering and software engineering. Due to the different cultures different levels of model abstraction and different formalisms are used by these different disciplines during the development process. Furthermore, different tools are employed for the development of these systems, as the involved disciplines are familiar with different domain specific tools. To handle such a multitude of modeling paradigms and tools support for an efficient approach, which combines the different models and tools is required. In this paper we show that our Mechatronic UML approach fulfills the needed requirements of a multi-paradigm modeling approach for reconfigurable mechatronic systems and moreover we show that this approach addresses the verification and code generation, which are important and critical aspects during the development process of these systems. Therefore, these aspects should be considered in a multi-paradigm modeling approach.

Keywords: Multi-Paradigm, Mechatronic Systems, Meta-Model, Modeling, Verification and Validation, Code-Generation

1 Introduction

Nowadays, reconfigurable mechatronic systems have reached a complexity that requires a model-driven development approach to ensure a correct and competitive realization. Due to their nature, reconfigurable mechatronic systems have strong dependencies between software engineering and control engineering. The system consists of continuous paradigms, e.g. differential equations, and discrete paradigms, e.g. statecharts, which have dependencies to realize the often required hybrid behavior of mechatronic systems. Therefore different paradigms are needed to be combined as for the whole system domain specific expertise is needed to realize a correct and competitive realization. Looking at the whole

development of mechatronic systems we have to look not only at the modeling but also at verification and validation techniques, like formal verification, and at the code-generation, which need to be combined.

If we look in more detail at the multi-paradigm development approach of mechatronic systems and further consider Mosterman and Vangheluwe [1], who identified three orthogonal dimensions for multi-paradigm approaches, which are i) models of different abstractions, ii) different formalisms, and iii) meta-modeling, we can see that there is a strong relation between paradigms and formalisms and between paradigms and different models of abstractions. The latter one is needed to enable the efficient and correct development of systems due to the concentration on the relevant aspects of the system requirements by omitting platform specific aspects. E.g. the behavior is specified by statechart models and not on code level. The first relation is obvious as e.g. statecharts are based on a syntactic and semantic definition. Therefore it is obvious that the problem of the combination of the different paradigms is a problem of transforming a formalism into an other one or the combination of different formalisms. In Tabular 1 paradigms used in mechatronic systems are presented. In our contribution, we focus on the relation and their mutual influence of the continuous and event-based paradigms. We show in the following Sections the dependencies and transformations between these paradigms.

paradigm	continuous	event-based	dynamic structural adaptation
formalism	block diagrams	automata	story pattern
semantics	differential equations	timed automata	graph transformation systems

Table 1: Paradigms used in Mechatronic UML

Besides the multi-paradigm modeling problem the different domain specific tools need to be combined to achieve the expertise of the different disciplines and therefore a one tool approach, which applies the different formalisms and ensures consistency, is not preferable. Therefore a tool supported approach needs to maintain the domain specific tools yield a consistency problem.

In this paper, in contrast to prior publications ([2],[3], [4], [5], etc.), which describe specific parts of the Mechatronic UML approach, we show how Mechatronic UML addeses the issues of multi-paradigm modeling. The different aspects are shown by a running example, which is introduced in the next Section 2. Mechatronic UML is introduced in Section 3 and the underlying meta-model is introduced in Sectionsec:metamodel. After that the relevant techniques modeling (cf. Section 5), Verification and Validation (cf. Section 6), and Code-Generation (cf. Section 7) are introduced. In Section 8 we discuss the related work and we finish the paper with a conclusion in Section 9.

2 Application Example

To outline our multi paradigm approach, we employ an example, which stems from the RailCab[1] research project at the University of Paderborn. Autonomous shuttles are developed, which operate individually and make independent and decentralized operational decisions.

The shuttle's active suspension system and its optimization is one example for a complex mechatronic system whose control software we design in the following. The schema of the relevant physical model of our example is shown in Figure 1.

The suspension module is based on air springs, which are damped actively by the displacement of their bases and three vertical hydraulic cylinders, which move the bases of the air springs via an intermediate frame – the suspension frame. The vital task of the system is to provide the passengers a high comfort and to guarantee safety when controlling the shuttle's car body. In order to achieve this goal, multiple feedback controllers are used with different capabilities in matters of safety and comfort [6].

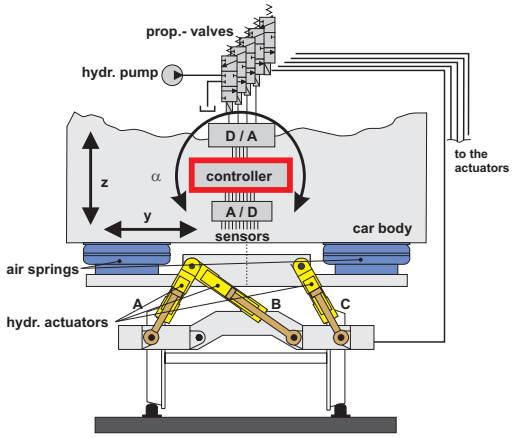


Figure 1: Scheme of the suspension module

3 Mechatronic UML

The Mechatronic UML approach enables the development of complex mechatronic systems [4]. Components and patterns can be employed to model the architecture and real-time coordination behavior. An embedding of continuous blocks into hierarchical component structures permits to integrate controllers into the component model. The component and pattern definition are sup-

[1] http://www-nbp.upb.de/en/index.html

ported by the Fujaba Real-Time Tool Suite[2] while the blocks can be specified with CASE tool CAMeL.[3]

Discrete behavior of the components and patterns is specified by real-time statecharts [5] or their hybrid extension hybrid reconfiguration charts [7]. The provided embedding concepts enable the specification and modular verification of reconfiguration across multiple components [3].

The MECHATRONIC UML approach also supports model checking techniques for the real-time processing at the level of connected mechatronic systems. By supporting a compositional proceeding for modeling and verification of the real-time software [2], the approach avoids scalability problems to a great extent.

Mechatronic systems typically are characterized by a high degree of complexity due to the strong cross-coupling of the involved different engineering disciplines. This complexity originates from the large number of couplings on various levels of the contributing elements and components, coming from different disciplines.

Typically, the development process of a mechatronic system can be represented by the well-known V-model. During the design process, the V-model is cycled completely or partially, depending on the individual requirements. Beside the V-model there are a couple of other development processes for the development of mechatronic systems. In all those common models, modeling (cf. Section 5), verification (cf. Section 6), and code generation (cf. Section 7) can be identified as the main paradigms.

4 Meta-Model of MechatronicUML

The core of our MECHATRONIC UML approach is the meta-model. The specific characteristic is that all above mentioned formalisms are combined by only one meta-model which becomes possible by a well defined union of all formalisms.

In Figure 2 a cut out of the MECHATRONIC UML meta-model is depicted. Both, the time-continuous behavior and structure as well as the continuous control behavior and structure are integrated in one meta-model. This is reflected in the Figure by *ContinuousComponent* (green borderd classes) and *DiscreteComponent* (blue bordered classes). Furthermore, the hybrid classes (HybridComponent, HybridPort, HybridComponentInstance, HybridPortInstance) combine both, the continuous and discrete parts.

5 Modeling

In the context of mechatronic systems typically a superior state diagram embeds continuous behavior in form of feedback-controllers, which leads to a hybrid system. Furthermore, in complex mechatronic systems the feedback-controllers could be exchanged during runtime, which leads to a reconfigurable system.

[2] http://www.fujaba.de/
[3] http://www.ixtronics.de/English/indexE.htm

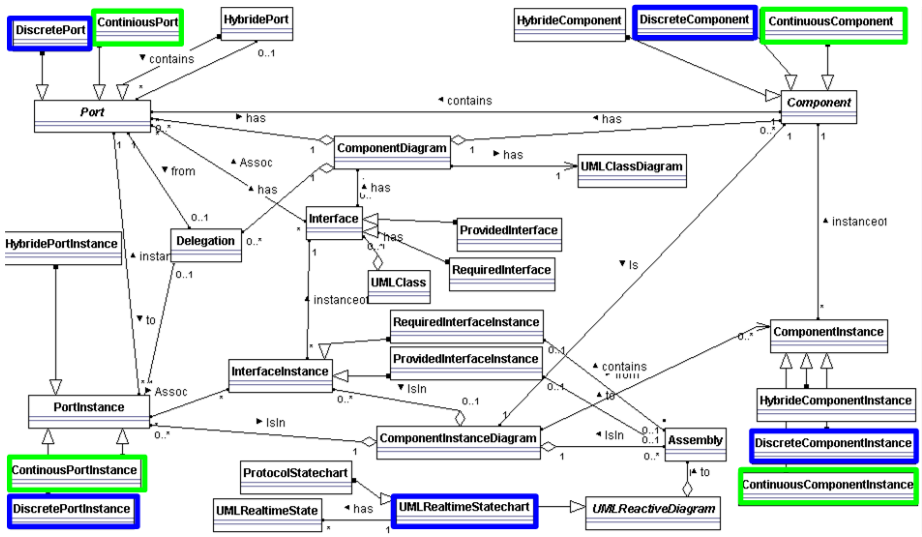


Figure 2: Cut out of MECHATRONIC UML meta-model

A modeling approach therefore have to support distributed, hybrid, and reconfigurable systems with real-time behavior, which should be verifiable and implementable as discussed in Section 6 and Section 7.

Multi-paradigm modeling is a prerequisite for a successful multi-paradigm development approach. We therefore show how the here considered domain specific approach ensure a multi-paradigm modeling approach of the aforementioned different paradigms like the continuous and discrete one. To order the different dependencies, we discuss in the next subsections the different aspects with respect to the structural and behavioral view.

5.1 Structural View

The architecture of the system is based on distributed, interconnected components. The components are based on UML 2.0 components but additional we distinguish between different types of components (see Figure 2). We distinguish between (discrete) components, hybrid components, and continuous components (block diagrams). As required in the introduction of the superior Section, components can embed other components. Additionally we can distinguish between component types and component instances.

Figure 3 shows an instance view of the component structure of the suspension system. There it is shown that a hybrid Monitor component embeds continuous Sensor and Storage component and moreover a hybrid BodyControl component, which embeds three different controllers (not shown in the Figure). Based on the availability of the information of the Storage, which stores track information of other shuttle, which are communicated from the registry, and the Sensor the BodyControl can switch between different controller based on the available

information. Besides the shown embedding of the different paradigms additional a more abstract view in form of pattern are also supported as shown in the Figure. A pattern in our case consists of port roles (MonitorRole, RegistryRole) and a connection between the roles. Besides the structure, a pattern consists also of a behavioral description (cf. Section 5.2).

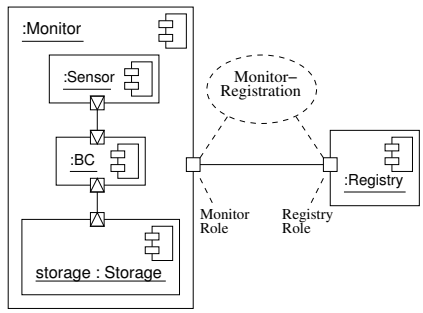


Figure 3: Structural description of the suspension system

5.2 Behavioral View

In the behavioral view the behavior of a pattern and of a single component is specified. As shown in Figure 3 the interaction between two components is specified by a pattern. The behavior of a pattern is described by specifying the behavior of the port roles. Figure 4 shows the behavior of the Monitor component. The lower And-State shows the behavior of MonitorRole and the upper And-State shows the behavior of the superior behavior of the Monitor component by embedding the Sensor controller (continuous component), Storage controller, and different instances of the body controller. As shown in Figure 5(a) the BodyControl statechart embeds in state Robust the Rob controller, in state Absolute the Abs controller, and in state Reference the Ref controller. The fading between controller is considered by so called fading transitions, which contain the relevant information in form of time needed for the fading (bold transitions in Figure 5(a)). An embedding of components is critical, as we have to guarantee a structural and semantically correct embedding (cf. 6). To abstract from irrelevant information, we introduce Interface Statecharts (cf. 5(b)). Interface Statecharts abstract from the embedded controllers and fading functions as for the correct embedding only the discrete real-time behavior is important (cf. 6).

Summarized, we have seen that besides the structural embedding of the continuous paradigm also the block diagrams are embedded in the behavioral description by Hybrid Statecharts as block diagrams describe the structure and the behavior (cf. Figure 2, HybridComponent, HybridPort, HybridComponentInstance, HybridPortInstance). Furthermore different abstractions are introduced to reduce the complexity and concentrate on the relevant aspects. To support the whole approach we have combined or linked the different modeling paradigms of the

control engineering and software engineering to enable modeling systems of reconfigurable mechatronic domain.

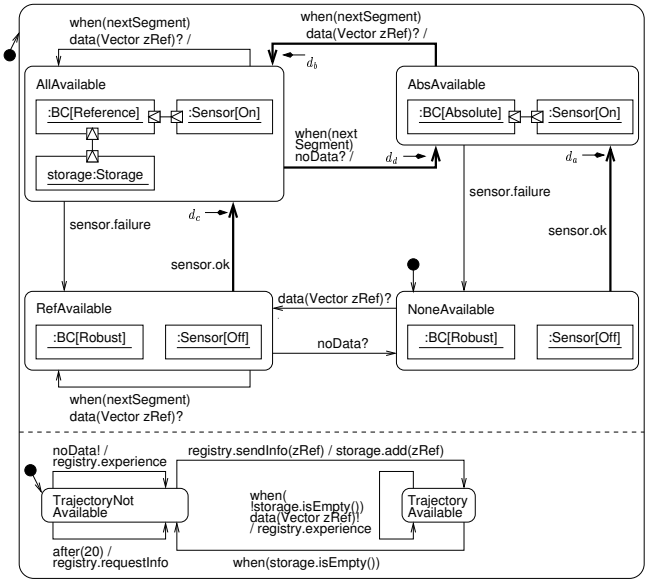


Figure 4: Behavior description of the Monitor

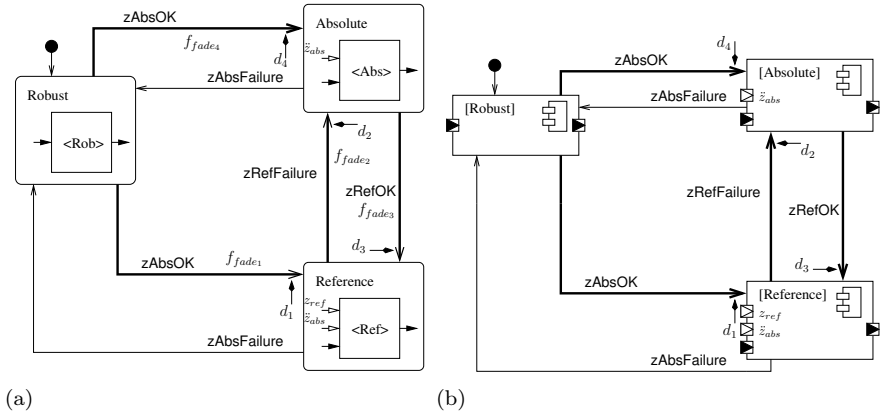


Figure 5: BodyControl (5(a)), and the interface state chart (5(b))

6 Verification and Validation

In this section we discuss the verification and validation phase of the development process of mechatronic systems. Therefore we look at formal verification

and show the transformation of the modeling formalisms to the (model) formalism of the formal verification.

In Figure 6 a sketch of the compositional and modular verification of our Mechatronic UML models is depicted. For the outlined Mechatronic UML approach, two specific verification tasks for the resulting systems are supported. At first the real-time coordination of the system, which is modeled with components and connectors are only interconnected by verified coordination patterns (cf. Section 5), can be verified using a compositional model checking approach [2]. To achieve this, we first use abstraction and furthermore, we transform the model formalism (real-time statecharts) to the formalism of the formal verification (time automata). Secondly, a restricted subset of the outlined hierarchical component structures for modeling of discrete and continuous control behavior can be checked for the consistent reconfiguration and proper real-time synchronization w.r.t. reconfiguration [3][8]. In addition, the second approach can be embedded into the first one by ensuring the verification steps that the synchronization with the subordinated components within the hierarchical structures does only result in a refined behavior such that the verification results for their interconnection via verified patterns still hold. The second approach performs the abstraction and formalism transformation as the first approach and additionally support the abstraction of the hybrid system.

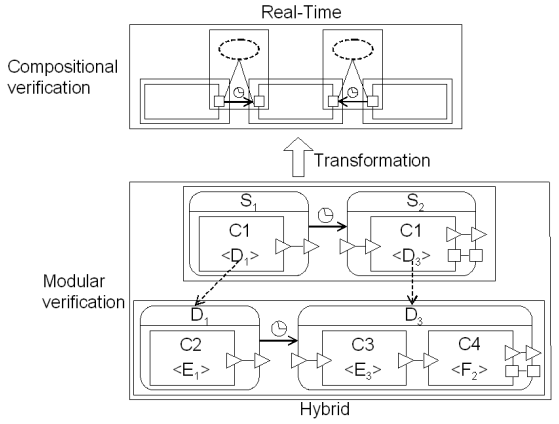


Figure 6: Overview about compositional and modular verification

7 Code-Generation

The Mechatronic UML approach applies the model-driven development approach to develop software systems at a high level of abstractions to enable analysis approaches like model checking [5] as shown in Section 6. Therefore, ideally, we start with platform independent models as shown in Section 5 to enable the compositional formal verification. Afterward, the platform indepen-

dent model must be enhanced with platform specific information to enable code generation. The needed platform specific information are based on a platform model, which specifies the platform specific worst case execution times and furthermore we add platform specific information like priorities, periods, and deadlines to the models. After specifying the platform specific information we can generate code from the models. We therefore apply three different models of abstraction, like the MDA approach, with conformable transformations.

The presented seamless approach is realized by the Fujaba Real-Time Tool Suite. As mentioned in Section 3 we integrate the Fujaba Real-Time Tool Suite with the CASE Tool CAMeL and therefore use the ability of CAMeL to generate code. As shown in Figure 7 both tools export hybrid components, which are integrated into a hierarchical model as a input for the binding tool. The export of the tools includes already the transformation of the model formalism to a code formalism, like C++. Afterward, the binding tool combine the inputs and therefore combines both formalisms.

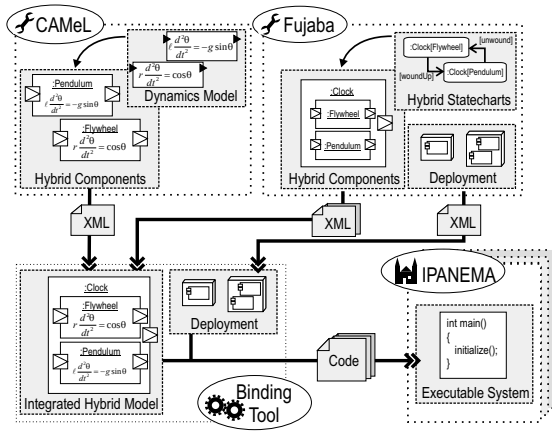


Figure 7: Tool Integration Overview [9]

8 Related Work

In [1] the field of Computer Automated Multi-Paradigm Modeling (CAMPaM) is introduced. This approach defines a domain-independent framework for multi-paradigm modeling that consists of three dimensions: (i) multi-abstraction, (ii) multiple formalisms, and (iii) meta-modeling. Due to the use of transformation techniques multiple formalisms are linked.

A multi-paradigm modeling approach within a Petri-Net framework for hybrid dynamic systems is presented in [10]. Based on the multi-paradigm modeling concept, modeling schemes of the hybrid system are separated, but combined in a hierarchical way through specified interfaces.

Ptolemy [11] supports different models of computation (semantic domains). It supports semantics for continuous time, discrete time, timed multitasking, etc. Furthermore it supports combination and integration of these semantic domains.

The de facto industry standard for modeling mechatronic systems with hybrid behavior is MATLAB/Simulink and Stateflow[4]. Here block diagrams and classical control engineering concepts have been integrated with a statechart dialect named Stateflow. The MATLAB/Simulink based tool CheckMate can be used to verify MATLAB/Simulink and Stateflow models.

The presented approaches support modeling and simulation but all these approaches fail in supporting a formal verification technique like model checking. As this must be considered in a multi-paradigm approach for complex technical systems like reconfigurable mechatronic systems, these approaches fail in supporting all levels of a multi-paradigm approach.

9 Conclusion

In this paper we presented MECHATRONIC UML a multi-paradigm modeling approach for reconfigurable mechatronic systems. We have shown that MECHATRONIC UML combines the different modeling paradigms of the software engineering and control engineering as both disciplines are inherently involved in the development of reconfigurable mechatronic systems. To reflect the problems during the whole development process, we address besides the modeling model-checking and code-generation with respect to the multi-paradigm approach.

Acknowledgements We thank Holger Giese for his comments on earlier versions of this article. This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft. Supported by University of Paderborn

References

[1] P. J. Mosterman and H. Vangheluwe, "Computer automated multiparadigm modeling: An introduction," in *Journal on Simulation*, vol. 80, pp. 433–450, SAGE, 2004.

[2] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake, "Towards the Compositional Verification of Real-Time UML Designs," in *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, pp. 38–47, ACM Press, September 2003.

[4] http://www.mathworks.com

[3] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp, "Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration," in *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pp. 179–188, ACM Press, November 2004.

[4] S. Burmester, H. Giese, and M. Tichy, "Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML," in *Model Driven Architecture: Foundations and Applications* (U. Assmann, A. Rensink, and M. Aksit, eds.), LNCS, pp. 1–15, Springer Verlag, 2005.

[5] S. Burmester, H. Giese, and W. Schäfer, "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*, Lecture Notes in Computer Science (LNCS), pp. 25–40, Springer Verlag, November 2005.

[6] T. Hestermeyer, P. Schlautmann, and C. Ettingshausen, "Active suspension system for railway vehicles-system design and kinematics," in *Proc. of the 2nd IFAC - Confecence on mechatronic systems*, (Berkeley, California, USA), December 2002.

[7] S. Burmester, H. Giese, and M. Hirsch, "Syntax and semantics of hybrid components," Tech. Rep. tr-ri-05-264, University of Paderborn, October 2005.

[8] H. Giese and M. Hirsch, "Modular verificaton of safe online-reconfiguration for proactive components in mechatronic uml," in *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers* (J.-M. Bruel, ed.), vol. 3844 of *Lecture Notes in Computer Science*, pp. 67–78, Springer Verlag, 2006.

[9] S. Burmester, H. Giese, and F. Klein, "Design and Simulation of Self-Optimizing Mechatronic Systems with Fujaba and CAMeL," in *Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany* (A. Schürr and A. Zündorf, eds.), vol. tr-ri-04-253 of *Technical Report*, pp. 19–22, University of Paderborn, September 2004.

[10] J.-S. Lee, M.-C. Zhou, and P.-L. Hsu, "A multi-paradigm modeling approach fpr hybrid dynamic systems," in *Proceedings of the 2004 IEE Conference o Computer Aided Control Systems Design, Taipei, Taiwan, Septemper 2-4*, pp. 77–82, IEEE Computer Press, September 2004.

[11] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng, "Overview of the Ptolemy Project," TechReport UCB/ERL M03/25, Department of Electrical Engineering and Computer Science, University of California, Berkeley, July 2003.

Using a Lattice of Coalgebras For Heterogeneous Model Composition

Jennifer Streb
Information and Telecommunication Technology Center
The University of Kansas
jenis@ittc.ku.edu

Perry Alexander
Information and Telecommunication Technology Center
The University of Kansas
alex@ittc.ku.edu

Abstract. System-level design is characterized by a need to bring together concurrent information from numerous domains to assess the impact of local decisions on global properties. As such, to support system-level design a language must minimally address heterogeneous specification, specification transformation and specification composition. We propose a semantics for defining these operations based on a lattice of coalgebraic system models. Within this lattice we can provide formal definitions for safety using Galois connections, specification transformation using functors, and specification composition using the classical sum and product operations from category theory. Embodied in the Rosetta specification language, this semantics has proved useful in assessing system level properties such as power consumption and security. This paper overviews the semantics and provides a simple example of its use.

Keywords: Heterogeneous Specification, Coalgebraic Semantics, Specification Composition

1 Introduction

The essence of system-level design is bringing together information from multiple, concurrent domains to assess global effects of local decisions. Thus, for any language or semantic system to address system-level design needs it must *support the representation of heterogeneous information* and *support composition of information across domains.* The Rosetta language and semantics [1, 2] are designed explicitly to support the needs of system-level design. It supports heterogeneous specification by providing a collection of *domains* that provide vocabulary and semantics for writing specifications and the *domain lattice* that supports transforming and composing information across domains. In this paper, we will eschew discussion of the main body of the Rosetta syntax, instead concentrating on the semantic infrastructure required to construct the domain lattice, transform, and compose models.

2 Background

Models in Rosetta are referred to as *facets* and are the fundamental unit of specification. Each facet represents one aspect or view of a multi-aspect system. Information such as component function, performance constraints, and structure are all represented using facet models. The key is that each facet represents a system from one perspective using a semantic basis appropriate for the information being represented. A complete system model composes facets representing multiple perspectives into a composite system model. The domain lattice provides support for these operations and a foundation for safety assurance while coalgebras form the semantic basis for individual models.

2.1 The Domain Lattice

Vocabulary and semantics for defining facets are provided by *domains*. Each domain provides to varying degrees units of semantic representation, a model of computation, and a domain specific modeling vocabulary. Ideally, a domain defines a collection of definitions that characterize a particular computation or modeling style. This may vary from simply unit-of-semantic definitions like the state_based domain that defines a simple stateful computation model to complex engineering domains like the digital domain that provides a complete semantics for writing digital system models.

When a facet is defined, it is declared as an element of a domain and inherits all of that domain's declarations. Formally, the new facet *extends* a domain to define a new model. For example, if a register facet is defined of type state_based, then the concepts of state, change and event are available as a built-in part of the specification vocabulary. In this way, a facet's associated domain defines its type and the type associated with a domain is the collection of consistent facets that can be defined by extending it.

When a new domain is defined, it is declared as a subdomain of an existing domain. Like facets, the new domain extends the original domain and inherits all of that domain's declarations. If a new discrete_time domain is defined as a subtype of state_based, then the notions state and change are inherited and refined within the new domain. The distinction between defining a domain and defining a facet is the domain can be further refined to define facets or other domains. When a facet is defined, it cannot be extended and defines a leaf in the domain.

The collection of domains and the extensions used to define them define a tree that is referred to in Rosetta as the *domain lattice*. The set of domains, D, together with the homomorphism relationships resulting from extension define a partially ordered set $(D, \Rightarrow)$. Join ($\sqcup$) and meet ($\sqcap$) can subsequently be defined as the least common supertype and greatest common subtype of any pair of domains. It can easily be proved that any domain pair will in fact have a least common supertype and a greatest common subtype. The **null** domain is the least domain in the collection and all domains inherit from it. **bottom**

is the greatest domain and inherits from all domains making it inconsistent. Specifically:

$$\forall f :: facet \cdot \mathsf{bottom} \Rightarrow f \wedge f \Rightarrow \mathsf{null}$$

Including **null** and **bottom** with the partially ordered set $(D, \Rightarrow)$ defines a lattice whose top and bottom elements are **null** and **bottom** respectively:

$$(D, \Rightarrow, \sqcup, \sqcap, \mathsf{null}, \mathsf{bottom})$$

2.2 Coalgebraic Semantics

The domain lattice organizes domains but says nothing about the semantics of domains and facets. A facet's underlying semantics are denoted by a coalgebra [3] defining observations on an abstract state, $\mathcal{X}$. The signature for a general coalgebra is:

$$\langle x, y, z, s \rangle :: \mathcal{X} \rightarrow T_x \times T_y \times T_z \times T_s$$

where x, y, z, and s are observations on $\mathcal{X}$ and T_x through T_s are the types of those observations. When s is treated as state, this signature has the form of a classic Rosetta facet coalgebra. For any observation, x, made relative to state, the associated type will be:

$$T_s \rightarrow T_x$$

a functional mapping from a state value to a value of the type associated with the observation. One particularly important observation is the next state given by `next(s)`:

$$T_s \rightarrow T_s$$

mapping one system state observation to another.

A facet's signature defines its associated coalgebra signature and its terms define the coalgebra function by placing constraints on individual observations. This denotation is relatively straightforward and will not be discussed in detail here. It suffices to understand that the parameters and declarations defined in a facet signature define observations on $\mathcal{X}$.

We choose coalgebras over their better known duals, algebras, due to the non-terminating and heterogeneous nature of the types of systems we model. Coalgebras are more natural than algebras for representing non-terminating systems. The inductive proof theory associated with algebras requires a base case or initial state that may not exist in many embedded systems. As stream transformers, coalgebras and their associated proof techniques are well equipped to deal with reactive, non-terminating embedded systems.

The heterogeneous nature of system-level specifications requires that multiple computation models be considered during modeling and analysis. In the coalgebra, $\mathcal{X}$ can be held abstract with no associated concrete type. In this case,

Rosetta states simply become observations of the abstract state making multiple simultaneous state observations possible. Furthermore, by defining relationships between states in different domains, one can relate information associated with one state observation to information associated with other state observations. This critical feature allows determination of when information observed in one domain impacts information observed in another.

3 Lattice of Coalgebras

The result of the domain lattice definition and the coalgebraic semantics of facets is a *lattice of coalgebras* that serves as the underlying Rosetta semantics. With this semantic basis, we can now put the domain lattice to work defining specification transformation and composition. Additionally, the lattice facilitates establishing the safety of such operations using Galois connections. Each of these is critical to supporting model heterogeneity and composition necessary for system-level design. Figure 1 shows a part of the lattice of domains defined for traditional Rosetta specifications.

Functors and products discussed in this section and homomorphisms discussed earlier are examples of reflective Rosetta operations making up the *facet algebra* used to compose specifications. Products compose specifications and functors transform specifications to define new specifications. Homomorphism is a relation over specification pairs. Other important facet algebra operations include equivalence (isomorphism), relabeling and instantiation.

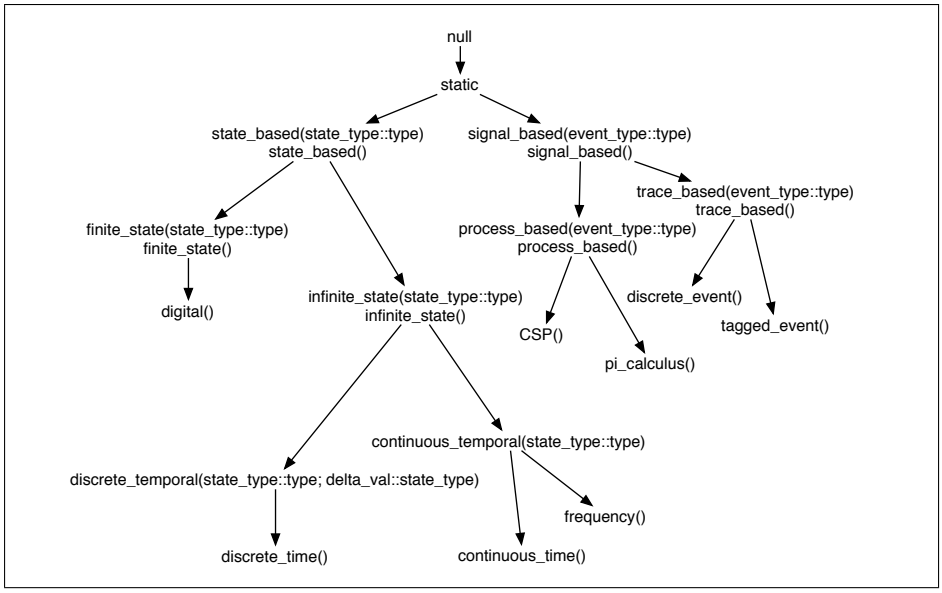


Figure 1: The Rosetta Domain Lattice

3.1 Functors and Specification Transformation

A *functor* in the domain lattice is a function specifying a mapping from one domain to another. The primary role of functors in the domain lattice is to transform a model in one domain into a model in another. Viewing each domain and facets comprising its type as a subcategory of the category of all Rosetta specifications, a functor is simply a mapping from one subcategory to another. Any model in the original category can be transformed into a model in the second. This corresponds to the classic definition of functors in category theory.

When defining domains by extension, two kinds of functors result. Instances of concretization functors, Γ, are defined each time one domain is extended to define another. Abstraction functions, A, are the dual of concretization functions and are known to exist for each Γ due to the multiplicative nature of extension. So, Γ instances move down in abstraction while A instances move up. In Figure 1 Each arrow moving from one domain down to another defines both an instance of Γ and A. However, A and Γ do not form an isomorphism because A is lossy – some information must be lost or A cannot truly be an abstraction function.

3.2 Safety and Galois Connections

Abstract interpretation [4] provides a capability for focusing analysis by eliminating unneeded detail from a specification. Among the most challenging problems in abstract interpretation is assuring that once the abstraction is performed the resulting model is faithful to the original. This is the notion of *safety* – assuring that when an abstraction is performed, the information retained is correct.

In the case of the Rosetta domain lattice, we need to verify the safety of functors moving specifications up and down the lattice. More specifically, we want to verify that by moving a specification or model between Rosetta domains we do not sacrifice correctness. Establishing a *Galois connection* [5] between domains in the lattice provides exactly this assurance.

A Galois connection (C, α, γ, A) exists between two complete lattices $(C, \sqsubseteq)$ and $(A, \sqsubseteq)$ if and only if

$$\alpha : C \rightarrow A \wedge \gamma : C \leftarrow A$$

are monotone functions that satisfy:

$$\gamma \circ \alpha \sqsupseteq \lambda c.c \tag{1}$$

$$\alpha \circ \gamma \sqsubseteq \lambda a.a \tag{2}$$

The two conditions above express that we do not sacrifice safety by going back and forth between the two domains although we may lose precision. For our purposes the notion of precision isn't important. We simply want to assure that by moving back and forth between domains we maintain a safe approximation of the original model.

Condition 1 states that abstraction (α) followed by concretization (γ) of a specification or model results in either the same specification or model, or one *more abstract* than the original yet still safe. Condition 2 states that concretization followed by abstraction of a specification or model will result in either that same specification or model, or one *less abstract* than it.

We have stated that extension of one domain to form another gives us a concretization function, Γ, that defines a homomorphism between domains. Because Γ is multiplicative, we are assured by nature of the lattice that an inverse, A, exists and can be derived from it. Thus, for any domain pair that is ordered by the lattice, we can define functors that move a specification between them.

With the domain lattice, A, Γ and the homomorphism, we can now define a Galois connection between any Rosetta domain, D_0, and any of its subdomains, D_1, as $(D_0, A_1, \Gamma_1, D_1)$. With the existence of the Galois connection we can now assure safety of any transformation between these two domains. Furthermore, the "functional composition" of two Galois connections is also a Galois connection [5]. Formally, if $(D_0, A_1, \Gamma_1, D_1)$ and $(D_1, A_2, \Gamma_2, D_2)$ are Galois connections then

$$(D_0, A_2 \circ A_1, \Gamma_1 \circ \Gamma_2, D_2)$$

is also a Galois connection. This is important because not only can we assure safety between any domain and its subdomain, but we can also assure safety of any transformation throughout the entire domain lattice.

The existence of the Galois connections are advantageous. They allow us to perform abstract interpretation with the certainty of the entire system still functioning as expected. They also allow multiple perspectives of a specification with the assurance of safety throughout. Additionally, we are guaranteed safety of any transformation within the entire Rosetta domain lattice due to the ability to functionally compose Galois connections.

3.3 Specification Composition

Heterogeneous specification truly becomes useful only when specifications can be combined to understand interactions. The domain lattice as presented thus far supports writing specifications using multiple models of computation and satisfies our goal of heterogeneity. Looking at the domain lattice from a categorical perspective enables using standard product and sum operations to perform composition.

The primary specification composition mechanisms in the Rosetta semantics are the *product* and *pullback* constructions [6]. A specification product is simply a pair of specifications that simultaneously describe a system. Because the specifications simultaneously hold, they must be mutually consistent. Mutual consistency between specifications in different domains implies consistency among heterogeneous specifications – precisely a goal of system-level design.

How can two specifications from distinct domains ever be inconsistent? Specifically, if two domains are distinct, then properties defined in one domain cannot reference the other because there are no shared symbols. The answer

lies in functors used to move information from one domain to another and in the pullback used to define the product.

In the traditional formal specification literature where algebraic semantics dominate, the *coproduct* and *pushout* are the dominant specification composition constructions [6, 7, 8]. Traditionally, a pushout of specifications forms the union of two specifications where shared specification that is jointly constrained in both specifications. With coalgebras, the product is the appropriate composition operator as we are looking for an interaction.

Formally, Given two models A and B the product is formed from the disjoint combination of A and B. As the composition is disjoint, there is no possibility of interaction. A pullback is a special construction for forming a product where each element is derived from a common specification, C. The elements of C are shared between specifications – when properties from A and B refer to elements of C, they are the same element. Properties placed on symbols of C from each specification mutually constrain C and A and B are no longer orthogonal.

4 Application Methodology

With semantic elements in place, we can outline a methodology for their application in a specification process. We start by defining specifications for system facets of interest. Functors defined by homomorphisms in the domain lattice are then used to move specifications to appropriate domains for composition and analysis. We then compose specifications using facet product operations. Finally, we verify the consistency of resulting specifications using simulation, theorem proving and model checking, and synthesize specifications into implementations using synthesis and compilation techniques. The resulting methodology is both effective and mathematically sound due to the Galois connections defined over the domain lattice.

4.1 Define Facet Specifications

The initial specification task is selecting specification domains for each relevant system requirements model and defining facet specifications for those models. A classic example from the Rosetta literature we use here is power-aware design where implementation fabric selection is a design decision made by understanding the interaction between functional requirements and power constraints. Example specifications are shown in Figure 2.

To effectively select domains and write specifications, the goals of the system analysis processes must be known. One must begin with the end in mind regardless of the system analysis methodology. Rarely does any system design process, specification or otherwise, yield anything useful that was not planned from the beginning.

```
facet power
   (o :: output top;
    leakage, switch :: design real)
    :: state_based is
  export power;
  power :: real;
begin
  power' = power + leakage +
           if event(o) then switch
              else 0
           end if;
end facet power;
```

```
facet interface function
    (i :: input real; o :: output real;
     clk :: in bit
     uniqueID :: design word(16);
     pktSize :: design natural)
     :: discrete_time is
  uniqueID :: word(16);
  hit :: boolean;
  bitCounter :: natural;
end facet interface function;
```

Figure 2: Rosetta specification fragments defining power consumption and functional models for a TDMA unique word detector. Due to space constraints, only the interface is shown for the functional model.

4.2 Transform Specifications for Analysis

Moving specifications using functors accomplishes two basic tasks: (i) moving a specification to an analysis domain; or (ii) moving a specification to a domain for composition with another specification. In the former case, specifiers move definitions from descriptive domains to new domains better equipped for analysis. In the latter case, specifiers move definitions to new domains where specification composition yields new, more detailed specifications. The Galois connections established over the Rosetta domain lattice guarantee the safety of abstraction and concretization functors.

Specifications for our example component exist in different domains that could be composed immediately using the product operation. In this case however, a more accurate performance prediction can be made if the specifications are composed in the same domain. Thus, we have three options: (i) Move the power specification to the discrete_time domain; (ii) move the functional specification to the state_based domain; or (iii) move both specifications to a common, intermediate domain. For this example, we choose to move the power specification into the discrete_time domain by applying a concretization functor. This decision is motivated by the existence of a discrete_time simulation environment exists that can be used to analyze the resulting specifications. The built-in concretization function for moving state_based specifications to discrete_time, gamma, is used for this task:

```
gamma(power());
```

The beauty of using gamma functions defined by the domain lattice is that if the extension between domains used to form the concretization function is consistent, gamma exists, alpha exists and the Galois connection assures their soundness. Thus, when moving the power specification above, we are certain

that the resulting specification in its new domain will be sound with respect to the original specification.

4.3 Compose Specifications

After transforming specifications into appropriate domains, the facet product is used to compose specifications. The specification product is formed from the power specification in the discrete_time domain and the original function specification. This new product facet is defined in Figure 3 using the application of gamma and the product operation.

```
facet power_and_function
    (i :: input real; o :: output top; clk :: in bit; uniqueID :: design word(16);
     pktSize :: design natural; leakage, switch :: design real ):: discrete_time is
  gamma(power(o,leakage,switch))
  * function(i, o, clk, uniqueID, pktSize);
```

Figure 3: Creating the composite specification by forming the product of the functional specification with the application of gamma to the power specification.

The product does far more than simply pair the specifications. Both specifications inherit a definition of time from the discrete_time domain. Specifically, a time value (t), quanta (delta) and next time function (next) are defined in discrete_time.

The product treats the discrete_time domain as a shared specification among the power and function models. The specification objects that t, delta and next refer to are shared between the specifications. Edges that indicate state change and power consumption are common to both components implying that processing in the functional specification results in power consumption in the power model. Any property defined on these items in one specification must be consistent with definitions in the other – they are literally shared between the specifications. Other symbols remain orthogonal, but when referenced in properties relating them to shared symbols they are indirectly involved in shared properties across domains.

4.4 Verification and Synthesis

With the composite model constructed, verification and synthesis are performed to predict system behavior and generate system components. Any system with a semantics compatible with the resulting Rosetta model can be used for analysis. In this case, simulation is performed by defining a domain associated with the simulator and using a functor to make the transformation. To preserve soundness, a verification obligation remains to show that simulator behavior is consistent with the domain describing it. Although non-trivial, this analysis is performed once per tool.

5 Related Work

Possibly the most visible work in heterogeneous modeling is the Universal Modeling Language (UML) [9, 10, 11]. Initially developed for object-oriented software systems, UML has expanded to digital hardware, embedded software and most recently system-level modeling. UML employs a collection of diagramming techniques whose semantics are customised using *profiles* specific to different domains.

UML meta-models provide additional semantics for heterogeneous systems that has been exploited for domain specific tool development and model-integrated design [12]. The model-integrated approach reflects our approach to model refinement and abstraction as the central features in design synthesis and analysis respectively. The model-integrated approach uses UML as its modeling language, although like the coalgebraic semantics presented here it should not be limited to UML models.

Viewpoints [13, 14, 15, 16] is a software specification technique where multiple perspectives of a software system are recorded. Viewpoints are less formal than Rosetta and focus primarily on software systems. However, interaction between models searching for inconsistencies has been explored extensively giving Viewpoints a similar system-level focus[17, 18, 19].

An alternative approach using operational modeling is the Ptolemy [20, 21] project. Ptolemy (now Ptolemy Classic) and Ptolemy II successfully compose models using multiple computation domains into executable simulations and actual software systems. Ptolemy II introduces the concept of a system-level type that provides temporal information as well as traditional type information. Specifically, the temporal characteristics of a type become a part of its description. Like Rosetta, Ptolemy II uses a formal semantic model for system-level types. Unlike Rosetta, Ptolemy models are executable and frequently used as software components.

6 Discussion

This paper overviews the approach to multi-paradigm specification embodied in the Rosetta specification system. We began with the assertion that Rosetta facets would be denoted as coalgebras, organized around domains situated in a lattice defined by homomorphism. Using the lattice as a basis, we discussed how homomorphisms define a Galois connection that assures the safety of information across specification transformations. Using the lattice and coalgebra semantics, we discussed how functors are used to move specifications between domains and how products are used to compose specifications in a well-defined, controlled manner.

Although space prevents discussing details of Rosetta or the Rosetta specification system, this overview should motivate further study of the lattice-of-coalgebras approach. We assert that the approach has potential beyond the

Rosetta specification system and our initial results in digital design, power-aware design and security suggest broad applicability.

References

[1] P. Alexander and C. Kong, "Rosetta: Semantic support for model-centered systems-level design," *IEEE Computer*, vol. 34, pp. 64–70, November 2001.

[2] P. Alexander, *System Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.

[3] B. Jacobs and J. Rutten, "A tutorial on (co)algebras and (co)induction." EATCS Bulletin 62, 1997. p.222-259.

[4] P. Cousot, "Abstract interpretation," *ACM Computing Surveys*, vol. 28, pp. 324–328, June 100.

[5] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer-Verlag, 2005.

[6] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. EATCS Mongraphs on Theoretical Computer Science, Berlin: Springer–Verlag, 1985.

[7] D. R. Smith, "Constructing specification morphisms," *Journal of Symbolic Computation*, vol. 15, pp. 571–606, 1993.

[8] D. R. Smith, "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 1024–1043, 1990.

[9] T. U. Group, *UML Metamodel*. Rational Software Corporation, Santa Clara, CA, 1.1 ed., September 1997. http://www.rational.com.

[10] T. U. Group, *UML Semantics*. Rational Software Corporation, Santa Clara, CA, 1.1 ed., July 1997. http://www.rational.com.

[11] A. Evans and S. Kent, "Core meta-modelling semantics of UML: The pUML approach," in *Proceedings of UML 99*, October 1999.

[12] A. Misra, G. Karsai, J. Sztipanovits, A. Ledeczi, and M. Moore, "A model-integrated infomration system for increasing throughput in discrete manufacturing," in *Proceedings of The 1997 Conference and Workshop on Engineering of Computer Based Systems*, (Montery, CA), pp. 203–210, IEEE Press, March 1997.

[13] A. Finkelstein, S. Easterbrook, J. Kramer, and B. Nuseibeh, "Requirements engineering through viewpoints," tech. rep., Imperial College, Department of Computing, 180 Queen's Gate, London SW7 2BZ, 1992.

[14] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in system development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, pp. 31–58, March 1992. World Scientific Publishing Co.

[15] S. Easterbrook, "Domain modeling with hiearchies of alternative viewpoints," in *Proceedings of the First International Symposium on Requiremetns Engineering (RE-93)*, (San Diego, CA), January 1993.

[16] J. C. S. do PradoLeite, "Viewpoints on viewpoints," in *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pp. 285–288, 1996.

[17] S. Easterbrook and M. Sabetzadeh, "Analysis of inconsistency in graph-based viewpoints: A category-theoretic approach," in *Proceedgings of The Automated Software Engineering Conference (ASE'03)*, (Montreal, Canada), pp. 12–21, October 2003.

[18] S. Easterbrook and M. Chechik, "A framework for multi-valued reasoning over inconsistent viewpoints," in *International Conference on Software Engineering*, pp. 411–420, 2001.

[19] S. Easterbrook and B. Nuseibeh, "Managing inconsistencies in evolving specifications," in *Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE-95)*, (York, UK), pp. 48–55, IEEE Press, April 1995.

[20] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, vol. 4, pp. 155–182, April 1994.

[21] J. Davis, "Ptolemy ii - heterogeneous concurrent modeling and design in java," 2000.

Constructing Multi-Paradigm Modeling Methods based on Method Assembly

Motoshi Saeki
Dept. of Computer Science,
Tokyo Institute of Technology
Ookayama 2-12-1-W83-3,
Meguro, Tokyo 152-8552, Japan
saeki@se.cs.titech.ac.jp

Haruhiko Kaiya
Dept. of Computer Science,
Shinshu University
Wakasato 4-17-1,
Nagano 380-8553, Japan
kaiya@cs.shinshu-u.ac.jp

Abstract. Computer Aided Method Engineering (CAME) is a kind of computerized tool for supporting the processes to build project-specific modeling methods and their supporting tools. This position paper presents the extended version of our CAME tool and discusses how to apply it to the tool integration for multi-paradigm modeling methods.

Keywords: Meta Modeling, Computer Aided Method Engineering, Method Assembly, UML

1 Introduction

Computer Aided Method Engineering (CAME) is a kind of computerized tool for the support of building project-specific methods for software development and generating their supporting tools [3]. One of the easiest ways to build them is to adopt reuse technique. In this technique, we have a kind of database system, called method base, which stores reusable method portions, called method fragments or method chunks, and assemble them into a situational method that can fit to a development project. Method assembly can be one of the techniques to construct situational multi-paradigm modeling methods. The supporting tools for this newly built method can be automatically generated by using the CAME tool. This position paper presents the extended version of our CAME tool [8] and discusses how to apply it to the construction and the tool integration for multi-paradigm modeling methods.

2 Requirements to CAME

Our CAME tool gets a method description (a meta model) as an input and produces the CASE tools that support the described method. We can list up the requirements to the technique of describing methods and to our CAME tool for multi-paradigm methods as follows.

1. The method description technique shall have sufficient power to express various kinds of methods. In this paper, we focus on the software requirements analysis and design methods whose artifacts to be produced are explicitly defined. The examples of our targets are the diagrams whose syntax is defined and texts including structured natural-language sentences where the syntax of words is clearly defined.

2. The method description technique shall be able to specify constraints on products developed following a method. These constraints are significant to keep consistency on the products.

3. The method description technique shall be able to define how the generated CASE tools co-operates so as to support the multi-paradigm method effectively. In the multi-paradigm method, several supporting tools, each of which supports a single paradigm method included in it, are tightly combined and used. In addition, the tools that cannot be automatically generated such as code generation and model transformation should be combined easily.

4. The generated CASE tools shall have the function of version control. The model being developed is frequently changed by requests of customers and/or by the changes of the situational environment. Unlike usual version control techniques for text documents such as CVS and Subversion, the units of version control are not lines of the documents but the logical concepts of the models such as Class in a class diagram and State in a state diagram.

3 Overview of Our CAME tool

3.1 Overall Architecture of Our Tool

Our CAME tool is based on reuse technique similar to the other existing CAME tools such as Decamerone [3] and MetaEdit+ [5]. Reuse technique is characterized by using reusable method portions, called method fragments or method chunks, which can be extracted from several existing methods. Method fragments are stored in a specific database called method base, and a special engineer, called method engineer obtains suitable fragments from the method base and assemble them into a new and project-specific method. The overview of our CAME tool is shown in Figure 1. The method engineer uses a method editor to manipulate the method fragments from the method base and assemble them into a new method. The method editor is a kind of diagram editor and allows the method engineer to easily edit method fragments. The method description consists of a product description and a process description. Our CAME tool automatically generates from the product description, 1) a modeling tool (CASE tool) for supporting inputting and editing products, such as the editor of Class Diagram, i.e. diagram editors and 2) the schema of a repository. In the meanwhile, the process description is used to generate a Navigation Browser

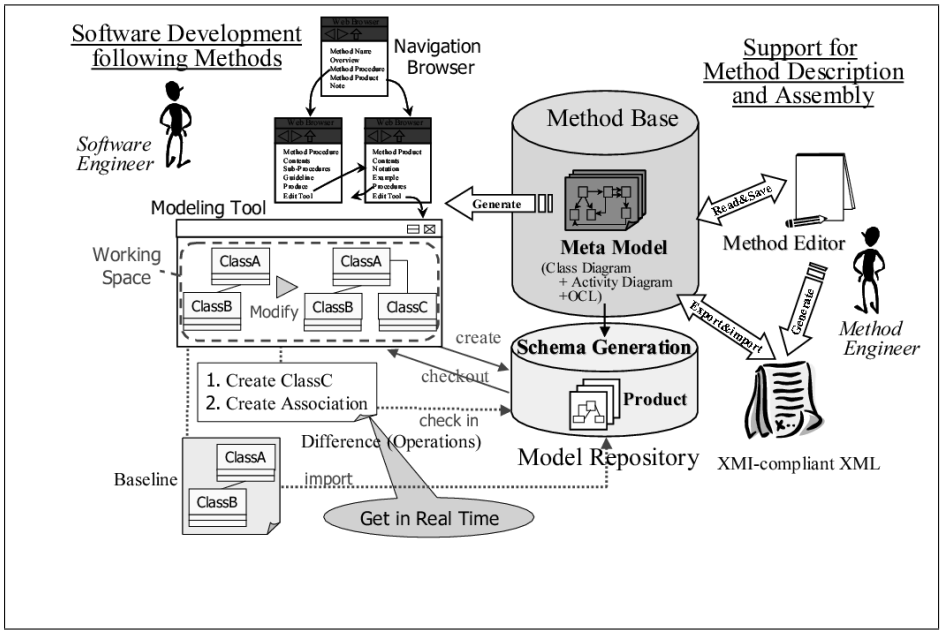


Figure 1: Overview of CAME

that guides software engineers' activities following the method. The generated modeling tools are not only for supporting inputting and editing products, such as the editor of Class Diagram, but also have version control mechanism.

The scenario of using the version control mechanism is illustrated as follows. A software engineer, i.e., a user of the generated modeling tool develops the first version of a model as a baseline, and imports it to the repository. He can check out any versions of the model that are stored in the repository, and edit them by using the editor. The editor gets the operation sequences on the model in real-time by monitoring the editor commands that he used. And he can check in the current version of the product by storing operation sequences as the difference to the repository whenever he wants to do. The details of this version control mechanism and the technique to generate it are discussed in [7].

3.2 Method Description Technique: Meta Modeling

In our method description technique, we adopt UML class diagram for specifying product descriptions of methods and UML activity diagram for process descriptions. In addition to these two facets, we should consider constraints on the products and on performing the activities of the process descriptions. To specify these constraints, we use OCL (Object Constraint Language).

The example of the method fragment is shown in a Figure 2 and it is the simplified version of class diagrams. In the figure, two sub windows PropertyEditor appear, the upper one is for specifying the properties of a product part and the lower one is for an activity. The method engineer defines the constraint

of the uniqueness of class names (no classes having the same name in a class diagram) with OCL through the upper ProperyEditor window.

The process part is described in hierarchical activity diagrams, and the figure illustrates the execution flow of the top-level activity “ConstructSimpleClassDiagram". The association “Produce" between the activity and the package “SimpleClassDiagram", a dotted arrow in the figure, represents the output product of the activity. The method engineer can use two associations “Produce" and “Consume" to specify output and input products respectively, and these associations are implicitly used to connect the generated CASE tools to the activity. For example, the generated tool for SimpleClassDiagram is automatically connected to the activity “ConstructSimpleClassDiagram" and when the method engineer starts this activity, the tool is automatically invoked, because the tool “produces" a SimpleClassDiagram. For an activity, the products specified with “Consume" are automatically displayed to assist in performing it. And, if no parts of the products specified with “Consume" are constructed yet, the method engineer cannot usually start the activity. The activity may be decomposed into lower level of activities such as “Identify Classes", “Identify Attributes" and so on. The method engineer can use any syntactic constructs of UML activity diagrams such as fork, join, branch etc. to define processes of the methods. There are several constraints to make a process description syntactically consistent and they are on activity diagrams. For example, an activity that has neither consumed nor produced products is inconsistent because it has no inputs or outputs.

We can set the conditions written in OCL in order to control tool invocations and the behavior of software engineers. These conditions are evaluated at a certain time, e.g. when the activity is entered or exits. If they are violated, the action specified by the method engineer is invoked, and this action can be specified as an executable Java program. In the lower window PropertyEditor of Figure 2, the condition (Product Condition) at timing “ProcessEntered" is a blank and it means “False". The action to be performed when the method engineer comes to the activity is “BootCASETool", and the CASE tool specified by him starts its execution. The method engineer can specify the tool name to be invoked by using the menu “BootCASETool" in Figure 2. In the example of the figure, the method engineer specifies the generated diagram editor “SimpleClassDiagram". Since the activity to “produce" a SimpleClassDiagram is only one and it is connected directly to the starting node (black circle) of the activity diagram, the editor is invoked when a software engineer starts this method. Although the method engineer can specify tool invocation by using “BootCASETool", more complicated processing on the constructed models may be necessary. Our tool provides Java API to access and manipulate the models, and the method engineer can develop Java programs to implement the complicated processing if necessary using the Java API. To get higher adaptability of the combination between activities and tools, the Java API is useful and this function is different from BPML [1]. The detail will be mentioned in the next section.

In addition to implementing and embedding the generation technique of version control mechanism, the newly extended parts of our tool from [8] is the description on how to combine the generated tools into an integrated tool by using UML activity diagram and OCL, as mentioned above. In [8], the process descriptions written in UML activity diagram were used for guiding model developers by showing what activities they should perform at next, as shown in Navigation Browser in Figure 1. On the other hand, in this extended version of the tool, we use UML activity diagram for specifying pre and post conditions of an activity and for defining actions to be performed within the activity. As a result, the method engineer cannot only specify the order of invoking the generated tools, but also check consistency of the model (an instance of the meta model) and transform the model for the later steps.

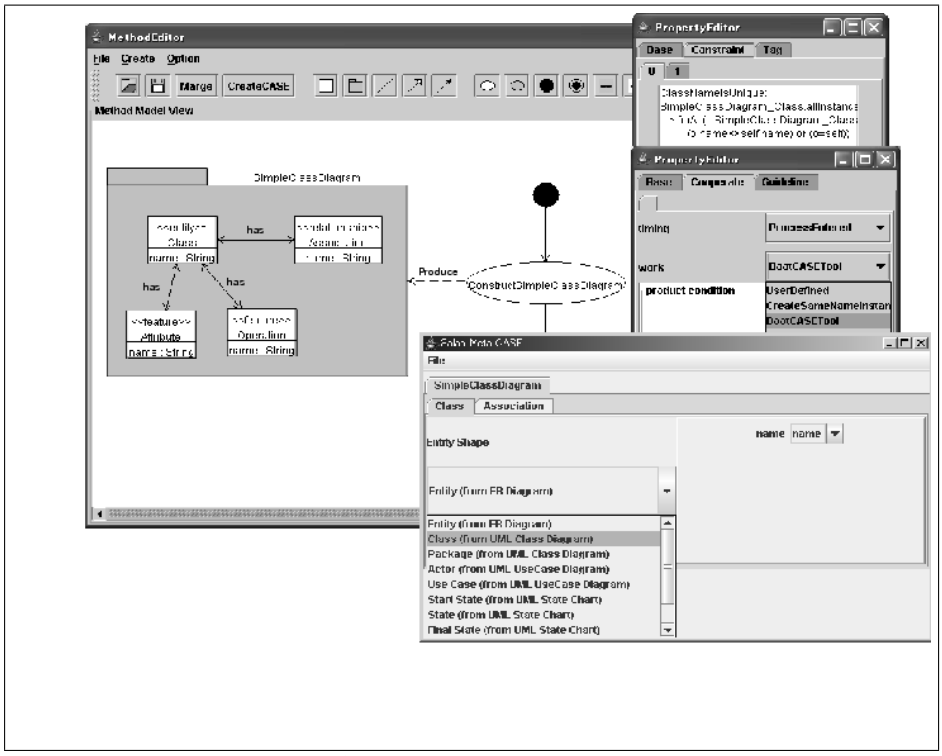


Figure 2: An Example of Method Fragments

3.3 Generating Diagram Editors

Our CAME is only for generating diagram editors from a product description, and it conceptually captures a product as a graph consisting of nodes and edges. To generate the editors, a method engineer should provide the information on which the elements in a method description can be represented with nodes or edges of the graph. He provides two types of information: one is the correspon-

dence of method elements to the elements of graphs, i.e. nodes, edges, texts within the nodes and texts on the edges, and another is notational information of the nodes and edges. Suppose that he tries to generate a class diagram editor from SimpleClassDiagram. The concept Class in the SimpleClassDiagram conceptually corresponds to nodes in a graph, while Association does to edges. He provides this information as stereotypes on the method elements. The readers can find the stereotypes "<<Entity>>" and "<<Relationship>>" attached to the classes in Figure 2. The former stereotype stands for the correspondence to a node and the latter to an edge. We have additional stereotype "<<Feature>>" to express the text data which are input from a dialog box in the generated modeling tool. In addition, he should specify which figures, say a rectangle, a circle, an oval and a dashed arrow, are used for expressing a method element on the editor screen. Basic graphical figures such as shapes used in UML diagrams are built-in and their drawing programs are embedded as Java classes into our CAME. A method engineer selects the figures out of these pre-defined built-in figures for the <<Entity>> components and <<Relationship>> ones, by clicking a menu item, as shown in the window "Salab Meta CASE" of Figure 2.

4 Multi-paradigm Methods by Method Assembly

Let's consider a simple scenario where a method engineer assembles the two method fragments StateDiagram (simplified version of usual State Diagram) and SimpleClassDiagram into a new method fragment Object Chart [4], which is drawn from [3]. In Object Chart method, a final product consists of a class diagram and state diagrams each of which corresponds to a class appearing in the class diagram. A state diagram specifies the behavior of the instances of the corresponding class, and its states can be annotated with attributes of the class. A software engineer develops a class diagram and then constructs a state diagram for each class.

Figure 3 illustrates the snapshot of this assembly task done by a method engineer. He gets the fragments StateDiagram and SimpleClassDiagram from the method base, and then customizes and combine these fragments for assembling them. For example, in the product part, he creates the new association "has", which combines StateDiagram and Class concept in SimpleClassDiagram, while in the process part he connects the ending point of the activity for SimpleClassDiagram to the starting point of StateDiagram as shown in the activity diagram of Figure 3. After completing combining the method fragments, the method engineer newly specifies the constraints, if any, to keep consistency on an Object Chart. In an Object Chart, events in a state diagram must appear as operations in classes in a class diagram. This constraint can be represented with OCL and it is set in the field "product condition" of the activity ConstructStateDiagram with the PropertyEditor window, which is the topmost of the three window occurrences of Property Editor. The method engineer selects ProcessExited to specify when the condition is checked in the window. Whenever the software

engineer finishes the activity ConstructStateDiagram, the generated tool checks if all events in the constructed state diagram also appear in the class diagram as operations or not. After setting the condition, the method engineer makes the “work" field “ShowMessage" so as to warn the software engineer when the constraint is violated, as shown in the middle window of the three ProperyEditors. The third window, i.e. the bottom of the ProperEditor windows shows the result of the method engineer's selection of the menus in the property editor of “ConstructStateDiagram".

The connection of the two generated diagram editors, i.e. SimpleClassDiagram editor and StateDiagram editor can also be defined in the same way, i.e. selecting menu items of the property editors. The method engineer can select the timing from ProcessEntered, ProcessExited, ProducedCASEToolBooted (the specified tool is invoked) and ButtonPressed (the specified button on the tool is clicked), and the action from UserDefined, BootCASETool, CreateSameNameInstances (pasting the objects having the same names from the products developed by another method), ShowMessage and ShowCandidateName. Three occurrences of the property editor windows in Figure 3 illustrate the process to specify the product condition, the timing and the action.

In Object Chart method, a class diagram is connected to state diagrams, and it is possible to generate a part of the state diagrams from the class diagram. In this example, the method engineer develops a Java program to generate a state diagram with blank sheet for each class in the class diagram, and selects the UserDefined menu and the timing ProcessExited at ConstructSimpleClassDiagram activity so as to register his developed program. Our tool provides Java API so as to develop this kind of Java program easily as a plug-in.

Figure 4 is a snapshot of the CASE tool for Object Chart method, which has been generated from its method description shown in Figure 3. When a software engineer (a user of Object Chart method and this tool) goes to the next activity by clicking a right directed block arrow in the menu of Navigator in the top left window of Figure 4, the execution of the tool for the next activity, if it is specified as timing ProcessEntered, is triggered. The left directed block arrow is for going back to the last activity. The software engineer can click the activity in the activity diagram in the right window of Figure 4 so as to start the clicked activity. When the software engineer changes the currently performed activity, the actions specified as timing ProcessExited or ProcessEntered are checked and executed if possible. Note that the software engineer can take the only actions through the specified tools at the activity. In this sense, we do not explicitly specify the possible actions of software engineers in the activity and this is different from GME [6] and AutoFocus [9].

The software engineer developed a class diagram at first (version 1) by using the generated class diagram editor and modified it as a version 2. The version tree viewer appears in the middle of the figure, and he can check out any version of the class diagram by clicking an item of the version list displayed in the left area. After finishing the version 2 of the class diagram, he moved to the next activity and selected the class “Lift". He is currently constructing a state

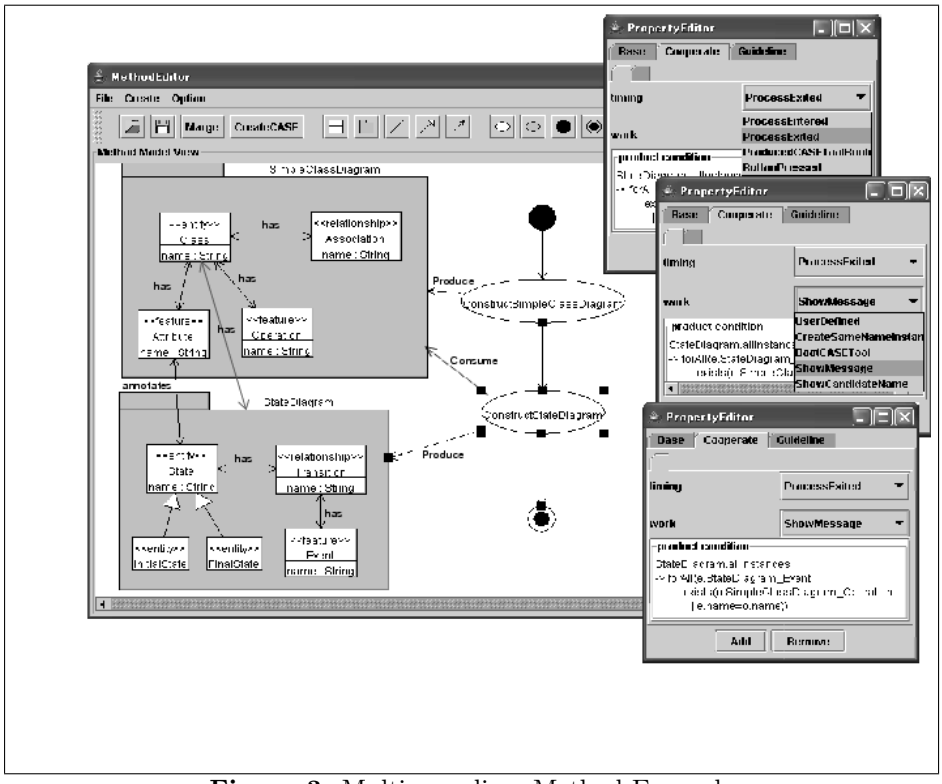


Figure 3: Multi-paradigm Method Example

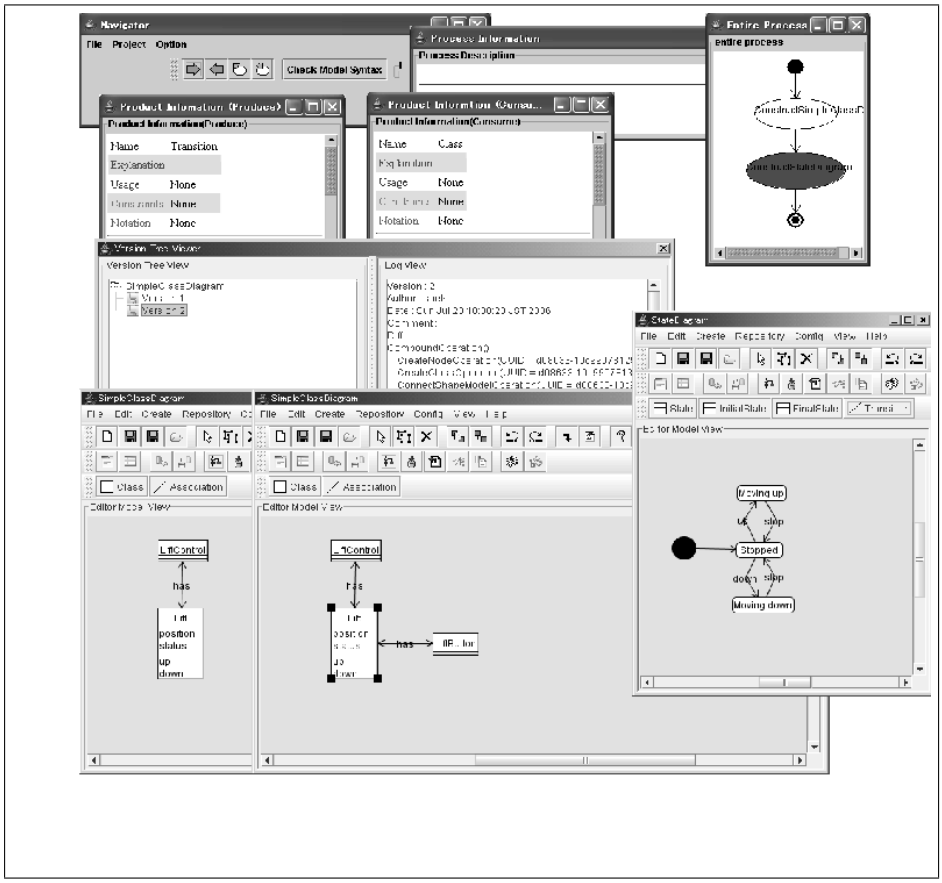


Figure 4: CASE Tool for Object Chart Method

diagram specifying the behavior of "Lift", by using a state diagram editor which has been automatically invoked.

Although we used non-hierarchical state diagrams for simplicity in this example, it is worthy of mentioning how to specify hierarchical diagrams with our tool. There are two alternatives, one is to introduce an association from a state to a package of "StateDiagram" in Figure 3, and another is an integration of two non-hierarchical state diagram meta models in a tree structure form. In the former alternative, the introduced association represents hierarchical links between state diagrams, and our CAME tool generates a single editor where all of the state diagrams are handled together. In the latter one, for each non-hierarchical state diagram, a diagram editor is attached and the corresponding editor is invoked following a hierarchical structure of the state diagrams, i.e. whenever a software engineer goes to a child state diagram. This is similar to Figure 3, where a class diagram and a state diagram is a parent and its child respectively.

5 Research Agenda

In this paper, we showed our computerized tool CAME and how to use to construct multi-paradigm modeling methods and their supporting tools. We have experienced in specifying meta models and in generating their editors for 12 basic diagrams (including 9 UML diagrams, UML action semantics diagram, Data Flow Diagram etc.), scenario analysis method (structured itemized text sentences), and goal-oriented analysis method (AND-OR goal graph). And we had constructed their well-known several integrations such as Class Diagram + State Diagram, and more practical methods such as OMT, Shlaer-Mellor, Scenario-Use Case method and etc. Shlaer-Mellor method was the most complicated in our case studies and was composed of three modeling methods; information, process and state models. Its meta model contains 24 classes in its product description and 18 activities in its process part. Through these experiences, we could realize the research issues as follows;

- More Powerful Support to Build Multi-Paradigm Methods
  Meta model based assembly is just a technique from syntactic aspect. To get meaningful multi-paradigm methods, we should explore semantical aspect of method fragments, e.g. by using Ontology technique [2, 3].

- Version Control and Configuration Management
  Although our CAME can generate the tools having version control and consistency checking mechanisms, these mechanisms work on method fragment level. In the example of Figure 4, a software engineer can construct any versions of class diagrams and state diagrams independently and they are managed separately. However these diagrams should be combined consistently and we need the support of selecting the consistent combinations.

  We adopted forward difference approach to store any versions of a model in a repository, but this approach has more shortcomings rather than backward difference one, in particular performance of checking out the newest version. The reasons of the forward difference approach are 1) the easier technique to handle with branched versions because the baseline version is only one and 2) research interest to explore how easy this experimental version of our CAME could be implemented and how less performance it had. In fact, all of the editors that the CAME generates have Undo and Redo functions and it is not so difficult to switch the current version with backward difference approach.

- Support the Development of Java Programs for Tool Cooperation
  The UserDefined menu in a property editor forces a method engineer to develop Java programs for controlling tool cooperation. Typical controls of tools frequently appearing in multi-paradigm methods can be collected and catalogued to patterns, templates and a library in order to support the development of these programs.

- Support for Code Generation and Model Transformation
  The techniques for code generation and model transformation, e.g. their

algorithms and transformation rules etc. greatly depend on the adopted meta models. For example, the code generation from class diagrams is different from state diagrams. Thus our tool currently provides Java API to access the constructed models so as to generate new models and their fragments, and a method engineer should implement meta-model-specific code generation and model transformation by means of the Java programs that use the Java API. As for model transformation, our tool can transform the models into XMI-compliant XML documents that can be inputs to the existing Graph Transformation Tool such as AGG [10], and currently we are developing the combination of AGG with our tool.

- Generating Various Types of CASE Tools from a Method Description
  In this paper, we limited the generation to diagram editors. The other types of CASE tools such as analyzers, e.g. calculating product quality, verifiers and simulators should be considered. Furthermore we consider a product as a graph conceptually and it leads to the limitation of the variety of its notation. In the example of UML Sequence Diagram, it can be logically represented with a graph. However it is not so suitable that we draw directly a sequence diagram as a graph, because geometrical information such as sizes and positions of figure objects (lines, arrows, rectangle boxes) in a sequence diagram is important. That is to say, in a logical level, considering a graph is sufficient, but in a notational level, a graph, which represents connections of figure objects only, is not sufficient. Wide variety of notation is necessary to deal with different kind of CASE tools.

- Usability and Performance of OCL
  We found several difficulties in specifying constraints with OCL because OCL has not enough retrieval functions, in the sense that more efforts were necessary to specify complicated constraints and conditions. And also, the performance of OCL evaluator was an issue when the size of the model was larger.

References

[1] BPML. http://www.ebpml.org/bpml.htm.

[2] J. Bezivin and R. Lemesle. Ontology-based layered semantics for precise OA&D modeling. In *Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, pages 31–37, 1997.

[3] S. Brinkkemper, M. Saeki, and F. Harmsen. Meta-Modelling Based Assembly Techniques for Situational Method Engineering. *Information Systems*, 24(3):209 –228, 1999.

[4] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Trans. on Soft. Eng.*, 18(1):9 – 18, 1992.

[5] S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+ : A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Lecture Notes in Computer Science (CAiSE'96)*, volume 1080, pages 1–21, 1996.

[6] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Proc. of WISP'2001*, 2001.

[7] T. Oda and M. Saeki. Generative Technique of Version Control Systems for Software Diagrams. In *Proc. of the 21st IEEE Conference on Software Maintenance (ICSM'05)*, pages 515–524, 2005.

[8] M. Saeki. Toward Automated Method Engineering: Supporting Method Assembly in CAME. In *Engineering Methods to Support Information Systems Evolution (EMSISE'03 in OOIS'03)*. http://cui.unige.ch/db-research/EMSISE03/, 2003.

[9] B. Schätz, P. Braun, F. Huber, and A. Wisspeintner. Consistency in Model-Based Development. In *Proc. of 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03)*, pages 287–296, 2003.

[10] G. Taentzer, O. Runge, B. Melamed, M. Rudorf, T. Schultzke, and S. Gruner. AGG : The Attributed Graph Grammar System. *http://tfs.cs.tu-berlin.de/agg/*, 2001.

Think Global, Act Local: Implementing Model Management with Domain-Specific Integration Languages

Thomas Reiter
Information Systems Group (IFS)
Johannes Kepler University Linz, Austria
reiter@ifs.uni-linz.ac.at

Werner Retschitzegger
Information Systems Group (IFS)
Johannes Kepler University Linz, Austria
werner@ifs.uni-linz.ac.at

Kerstin Altmanninger
Department of Telecooperation (TK)
Johannes Kepler University Linz, Austria
kerstin@tk.uni-linz.ac.at

Abstract. In recent years a number of model transformation languages have emerged that deal with fine-grained, local transformation specifications, commonly known as *programming in the small* [13]. To be able to develop complex transformation systems in a scalable way, mechanisms to work directly on the global model level are desirable, referred to as *programming in the large* [26]. In this paper we show how domain specific model integration languages can be defined, and how they can be composed in order to achieve complex model management tasks. Thereby, we base our approach on the definition of declarative model integration languages, of which implementing transformations are derived. We give a categorization of these transformations and rely on an object-oriented mechanism to realize complex model management tasks.

Keywords: model integration, model transformation, model management, domain-specific languages.

1 Introduction

Model-driven development (MDD) in general aims at raising the productivity and quality of software development by automatically deriving code artifacts from models. Even though an immediate model-to-code mechanism can yield tremendous benefits, it is commonly accepted that working model-to-model mechanisms are necessary [23] to achieve integration among multiple models describing a system and to make models first-class-citizens in MDD.

In recent years, therefore, a number of model transformation languages (MTLs) have emerged, which allow to specify transformations between metamodels. Such transformations are defined on a fine-grained, *local* level, upon elements of these metamodels. Albeit the advantages that MTLs bring in terms of manipulating models, it is quite clear that defining model transformations on a local level, only, can pose substantial scalability problems. Similarly, [9] emphasizes the need for establishing relationships between macroscopic entities like models and metamodels, for instance for the coordination of various domain-specific languages.

There are already first approaches trying to alleviate the above mentioned problem from two different angles (cf. also Section 5). The first category adheres to a bottom-up approach, meaning that existing general purpose MTLs are extended for special tasks like model merging [14] or model comparison [21]. Furthermore, mappings carrying special semantics can be established between metamodels and further on be derived into executable model transformations [6].

The second category of approaches is top-down-oriented and falls into the area of model management, where relationships between models are expressed on a coarse-grained, *global* level through generic model management operators. The aim of model management is to ease the development of metadata intensive applications, by factoring out common tasks in various application scenarios and by providing generic model management operators for these tasks. The operators' generality allows to make assumptions about, e.g., algebraic properties of model management operations, but does not necessarily make any specific assumptions about the operators' actual implementations. For instance, Rondo [5] is an implementation of such a system, oriented towards managing relational and XML schemata.

It is our opinion that both, bottom-up and top-down approaches are valuable contributions and should be considered as potentially complementing each other, as opposed to be thought of as two sides of a coin. One of model management's main contributions is to provide a conceptually well-founded framework guiding the actual implementation of model management operators, for which the capabilities of increasingly more powerful MTLs can be leveraged.

Therefore, this paper represents early work in drafting an approach that tries to build on the strengths of both paradigms. On the one hand, the model management rationale to make models first-class-citizens and to achieve complex model management tasks by assembling global operations on models, is followed. On the other hand, our approach relies on domain-specific languages (DSLs) developed atop general-purpose MTLs for locally handling fine-grained relationships between metamodels. The proposed approach resides in the context of the ModelCVS [18][17] tool integration project, which aims at integrating various modeling tools via metamodels representing their modeling language. Concretely, the problems that need to be solved are finding efficient ways to integrate various metamodels on a local level, and solve common problems, e.g., metamodel evolution, on a global level.

The remainder of this paper is structured as follows. Section 2 discusses the rationale behind our approach. Section 3 deals with the composition of model management operators and classifies different kinds of transformations. Section 4 goes into detail about how domain specific integration languages can be defined. Section 5 discusses related work and Section 6 summarizes our approach.

2 Rationale for our Approach

To better motivate the rationale underlying our approach, this section starts with an analogy referring to the definition of primitive recursive functions. Table 1 shows the various abstraction layers our approach is built on and introduces terms and concepts used throughout this paper. Referring to computability theory, using only the constant, successor, and projection functions, all primitive recursive functions, such as addition or subtraction operators, can be defined. Analogous to that, on top of existing *model transformation languages* residing on the local level, we define *integration operators* on the local composite level for handling fine-grained relationships between model elements. Algebraic as well as integration operators are then bundled up into sets representing algebras or *integration languages*, respectively. We refer to this level as intermediate, because the elements of algebras and integration languages act upon the local level, but are used to define transformations acting upon the global level. Hence, on the global level, complex functions and concrete realizations of model management operators are found. These algebras and languages are at a suitable level of abstraction and are commonly used to assemble algebraic terms or model management scripts [4]. After establishing a view across the abstraction layers, ranging from bottom-level MTLs to top-level model management scripts, we illustrate our approach in a top-down fashion in more detail.

Level	Natural Numbers	Example	Proposed Approach	Example
Global Composite	Terms	`power2(max(x,y))`	Model Mgmt. Scripts	`m'=translate(m.merge(m'))`
Global	Complex Functions	`power2(z),max(x,y)`	Model Mgmt. Operators	`Translation,PackageMerge`
Intermediate	Algebras	`{+,-,N},{*,/,N}`	Integration Languages	`FullEquivLang,MergeLang`
Local Composite	Operators	`+,-,*`	Integration Operators	`FullEquivClass,MergeClass`
Local	Base Functions	`succ(x),null()`	MTL Expressions	`ATLRule,OCLExpression`

Table 1: Analogy referring to the definition of primitive recursive functions.

Global and Global Composite. As depicted in Figure 1, we believe it is helpful to view the composition of complex model management operations as an object-oriented (OO) meta-programming task [2], where models are understood as objects and transformations as methods acting upon these "objects". Consequently, we think that an integral part of defining a metamodel should be to specify *integration behavior* in the form of transformations (1) that are

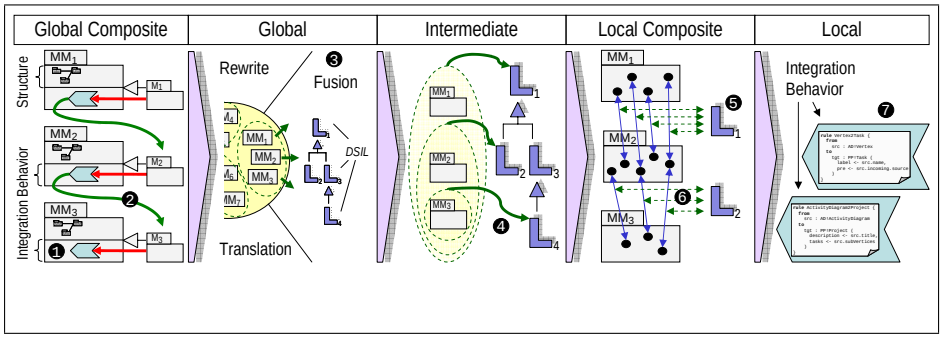


Figure 1: Illustration of our approach's abstraction layers.

tied to that metamodel (e.g., merging state-machines). The composition of transformations can then be facilitated by writing model management scripts in an OO-style notation, which invokes transformations on models (2) just like methods on objects. Transformations representing actual realizations of model management operators are defined by languages (3) which we refer to as *domain specific integration languages* (DSIL).

Intermediate. A DSIL consists of operators that enable to locally handle fine-grained relationships between metamodels and is formalized as a weaving metamodel [7]. The domain specificity of a DSIL stems from the fact that a DSIL can only be applied to certain kinds of metamodels (4). For instance, a *MergeLang* may be used to specify a merge for metamodels representing structures (e.g., class diagrams). As behavioral integration poses a very different challenge than structural integration [25], a merge on a metamodel representing some kind of behavior (e.g., business process), would have to be specified in a *FlowMergeLang*, whose operators are specifically aimed towards metamodels representing flows [24]. Efforts to formalize a metamodel's domain (e.g., by mapping metamodels onto ontologies [19]), could help to check whether a metamodel falls into the domain of a certain DSIL. From our point of view, this still poses an open research question and the applicability of a DSIL on a metamodel ultimately requires a user's judgement.

Local and Local Composite. An *integration specification* in a DSIL is a *weaving* model that conforms to its *weaving metamodel*, which is a certain DSIL's metamodel (5). A weaving consists of a set of typed links between elements of a model or a metamodel. The types of links represent different kinds of *integration operators* (6), whose execution semantics are defined through a mapping towards an executable MTL. Thus, an integration specification is finally derived into an executable model transformation (7).

Notably, our approach focuses on specifying integration between metamodels in a purely declarative way, as such a specification (which abstracts imperative implementations) is the basis for reasoning tasks like analysis or optimization.

3 Managing Models on a Global Level

This section discusses the two top-most layers of abstraction which have been previously introduced as *global* and *global composite*. The following subsection exemplifies transformation composition on the global composite layer through a model management script. Based on observations gained in the example, the global level is elaborated on in more detail by laying out a useful classification of transformations.

3.1 Model Manangement Scripts on the Global Composite Level

The following example deals with the merging of two domains represented by two metamodels, as depicted in Fig. 2. When these metamodels are merged, however, also their conforming models should be merged. We refer to such a model management task as an *exogenous merge*. A concrete application would be to merge previously modularized metamodels (e.g., a BPEL metamodel split into a structural and a behavioral part) or to extend a metamodel with a certain aspect (e.g., add "Marks" to a Petri-net metamodel) [20]. Throughout the example, however, for simplicity reasons and to emphasize the global perspective at this abstraction layer we will not go into detail about the makeup of the metamodels, which are simply referred to as A and B and their conforming models as a and as b, respectively.

There may be multiple ways to describe an *exogenous merge*. A straightforward way would be to program the whole task as one monolithic transformation in a general purpose transformation language. As already argued before, such ad-hoc approaches suffer poor scalability and reuse potential. Instead, a description of such complex tasks as a composition of global model management operations favors scalability and reuse: Firstly, one is not concerned with handling fine-grained relationships on the local model element level, and secondly, model management operations can be easily reused in order to assemble scripts for different tasks. Thinking of model management scripts as OO programs, as we propose to do, furthermore has the advantage that the code for this model management script does not need to be changed in order to work with other metamodels, as the actual transformations that are invoked, are *dynamically* bound depending on a model's metamodel.

Fig. 2 depicts the described setting and gives a listing of the according *exogenous merge* model management script. Details of the various steps in that script are discussed in the following.

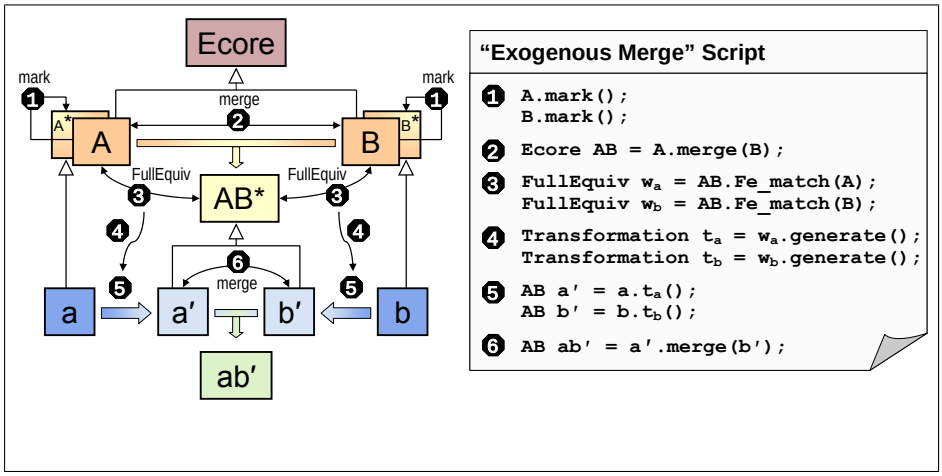


Figure 2: Model management script for exogenous merge.

In the first step (1) a *mark* transformation is run that tags all metamodel elements with a unique id by adding annotations. In the second step (2) a *merge* transformation is executed that unites the metamodels A and B as specified in the *merge* integration specification, for instance through overlapping the two metamodels on certain join points. This results in a new metamodel AB, which also contains the initially introduced markings. In the third step (3) a transformation creates a weaving between each of the original A and B metamodels and the newly created AB metamodel. A transformation creating such a weaving does a relatively easy job, as it can rely on the previously introduced traceability annotations to match model elements. The weavings created in our example comprise a certain integration specification, which in step (4) is derived into executable transformations, which are executed in (5) and migrate the models a and b towards models a' and b' that conform to the AB metamodel. Since these models now conform to the same metamodel, they can be overlapped in a *merge* transformation (6). We would like to mention, that also other ways of realizing traceability mechanisms exist, for instance through weaving a traceability aspect into a base transformation in an aspect-oriented fashion [16]. Embedding traceability information into a model through annotations, in our opinion has the advantage that a transformation producing a weaving can relatively easy create a trace weaving model. For further processing, the annotations could be easily pruned from the model.

3.2 Categorizing Transformations on the Global Level

After having discussed the composition of global model management operations, the following section will establish a better understanding of the transformations that were used in the previous example. However, this will not be done by discussing the behavior of these transformations in terms of how model elements are manipulated, as this is transparent on the global level and would differ for

different kinds of metamodels. Rather, the global level requires to put thought on what *kinds of transformations* are being employed.

Hence, we classify our approach's DSILs used to define actual transformations, into certain categories. These categories reflect recurring kinds of transformations prevalent in model engineering. Such a categorization favors the definition of modular and comprehensible transformations and creates a mindset where one can think of solving complex model management tasks through composition of such modular transformations, as exemplified in the previous subsection. Another advantage of this approach is that for every category a generic toolset can be built that allows to manipulate languages falling into a certain category. Transformations producing weavings can all share a tool like the Atlas Model Weaver [7], whereas translating transformations, for instance, can benefit from tooling to capture execution traces.

A similar distinction is made in the area of generic model management [4]. However, we allow the distinction between different categories according to the kind of input (IMM) and output metamodels (OMM) (cf. Table 2) that the transformations act upon, as opposed to focus on making assumptions about the behavior or algebraic properties of transformations.

Table 2 gives an overview by showing a category's input/output characteristics, example transformations, a reference to similar operators proposed in literature, and a function signature being representative for a category's transformations. To put each of the example transformations in a concrete context, we refer to the previously used traceability mechanism in more detail now. First, the *containsAnnotations* transformation is called to check whether a model is free of traceability annotations. If so, with *addTraceAnnotations* traceability annotations are added to all model elements. Next, *translateWithAnnotations* or *mergeWithAnnotations* is called that produces an output model in which the traceability annotations are migrated from source to target model elements. Then, *matchByAnnotations* is invoked which establishes a weaving model representing traceability links according to the annotations contained in source and target model. In a final step, this traceability weaving is input to the *createReverseTranslation* transformation which produces a round-tripping translation transformation.

Category	Arity	Output	Function Signature	Example	Operators in Lit.
Check	1	Prim. Type	`P p=check(M m);`	containsAnnotations	Check-property [12]
Rewrite	1	OMM==IMM	`M m'=rewrite(M m);`	addTraceAnnotations	Refactorings [17]
Translation	1	OMM!=IMM	`Mb mb=translate(Ma ma);`	translateWithAnnotations	ModelGen [3]
Fusion	2	OMM==IMM	`M m=fuse(M ma, M mb);`	mergeWithAnnotations	Merge [12]
Relation	2	Weaving	`W w=relate(Ma ma, Mb mb);`	matchByAnnotations	Match [3]
Generation	1	Transform.	`T t=generate(W w);`	createReverseTranslation	GlueCodeGen [11]

Table 2: Categories of transformations on the global level.

Check. The first category deals with transformations that map models onto primitive value ranges, like booleans or natural numbers. This kind of func-

tions allow to determine whether certain properties hold for models (consistency checks), or to evaluate certain criteria (e.g., number of inheritance relationships) of models.

Rewrite. This category encompasses transformations that modify a model but do not transform it into a model of another metamodel. This kind of transformations can be associated with editing or specialized refactoring operations [17], that do not require input from another model. An example language discussed later on is a language that allows to mark elements in a model with certain annotations.

Translation. A translating function maps concepts of one metamodel onto concepts of another metamodel and henceforth transforms a model conforming to one metamodel into a model conforming to another metamodel. A special case of a translating transformation would be if the source and target metamodels are the same, but nevertheless concepts are translated into other concepts. This would especially be the case when using UML, which, by means of stereotypes or tagged values offers a somewhat weaker mechanism than DSLs to represent concepts. Still we consider such transformations as part of this class, as the same translation language constructs can be of use, even though binding these needs some special effort.

Fusion. We classify a transformation as a fusion, if it takes two models as input and produces an output model taking into account each of the inputs. The input and output models thereby conform to the same metamodel. For instance, this class includes transformations that are usually associated with a merge or a diff [12], although domain specific realizations may potentially blend these two behaviors, by overlapping and clipping certain parts of the source models.

Relation. Transformations of this kind produce special kinds of models, which relate two other models. These models are referred to as weaving models [7] and consist of typed links between elements of left-hand side (LHS) and right-hand side (RHS) models. An example for a transformation creating a weaving could be carried out through a matcher, which heuristically establishes weaving links. Therefore, the creation of a weaving is often a task involving manual effort.

Generation. This kind of transformations generates other transformations. More precisely, they function as a compiler which turns weaving models into executable transformations. Typically this is either accomplished through a transformation whose target metamodel is the abstract syntax of a model transformation language or through a templating mechanism. It is important to note, that our view of a weaving is that a weaving model implicitly references its LHS and its RHS model, hence we omit these models in the above signature. Thus, we can still assume that the generation function has access to read the LHS and RHS models.

4 Integrating Models on the Local Level

The previous section has detailed the global composite and the global level. Hence, this subsection focuses on the remaining abstraction layers. As integration languages reside on the intermediate layer and this section makes use of a concrete example DSIL, the first subsection is dedicated to the *intermediate* level and to introducing the example language. The second subsection discusses the *local composite* level and discusses integration operators for the example DSIL. The *local* level is dealt with in the third subsection and focuses on the definition and extension of execution semantics for integration operators through a mapping towards MTL code.

4.1 An Example DSIL on the Intermediate Level

The abstract syntax of a DSIL is defined in a weaving metamodel [7], which is basically made up of meta-classes for the languages' integration operators. Furthermore, constraints are specified that enable to check whether a certain integration specification is valid. Such an analysis is comparable to static compile-time checking in traditional programming languages. In the following we will give an example for a basic language for the *translation* category. Due to space limitations we will not go into detail about languages of other categories, just as we are not claiming that the described integration operators are complete, as a precise definition is out of scope of this paper.

The setting for our example is depicted in Fig. 3, which shows a simple metamodel for activity diagrams (AD) as the LHS metamodel, and a Gantt-chart project plan (PP) metamodel as the RHS metamodel. An activity diagram consists of vertices and transitions in-between. A project consists of a number of tasks and every task has a reference to its previous task.

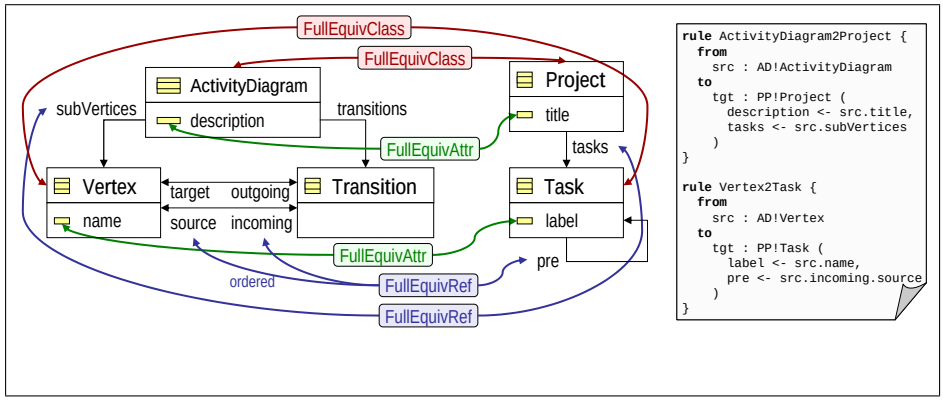


Figure 3: Example integration specification in the *FullEquiv* language.

The intention is to transform ADs into PPs in a semantics preserving way. Instead of programming the transformation directly, a DSIL is used to specify a mapping that denotes the translation of concepts of the AD metamodel onto

concepts of the PP metamodel. The code snippet on the right side of Fig. 3 shows the final transformation code that should be generated in an ATL-like[1] notation.

4.2 Integration Operators on the Local Composite Level

The DSIL used is the so called *FullEquivalence* language, which can be seen as a basic language for the *translation* category. It consists of three operators, namely *FullEquivClass*, *FullEquivAttr*, and *FullEquivRef*, which in a pair-wise manner link classes, attributes, and references, respectively. During the definition of a DSIL, it is important to define how its operators relate to each other. In our example, for instance, the *FullEquivAttr* and the *FullEquivRef* operators have to stand in the context of the *FullEquivClass* operator, as the assignment of values and the setting of references needs to happen in the context of the model elements which these attributes and references belong to. Such a relationship is defined through containment in the metamodel of the *FullEquivalence* language by making the *FullEquivAttr* and the*FullEquivRef* operators children of the *FullEquivClass* parent. Relationships not inferable from structure (e.g., precedence rules) can be specified in a constraint language. An example for a constraint that should be enforced is that an attribute in a target model element cannot be referenced by more than one *FullEquivAttr* operator having the same *FullEquivClass* parent, as this would lead to ambiguity concerning which source attribute should be used to set the target attribute.

4.3 Mapping Integration Operators onto the Local Level

After describing the operators, in the following example it is shown how a generating function can derive an implementation in the form of MTL code. Furthermore, we will exemplify the extension of an existing operator's semantics. The execution semantics are expressed through a function, mapping integration specifications expressed as weaving models onto executable transformations. This is either achieved through a template producing MTL code, or through a transformation creating a transformation program encoded as a model (higher-order transformation). However, writing transformations that produce transformation programs can be a daunting task. Thus, for better understandability, our explanation uses an example template language, which allows to see the output in bits of concrete syntax more intuitively.

Depending on what kind of transformation engine is used, the semantics of the resulting transformations are for instance formalized as abstract state machines [15] or as graph-based formalisms, such as triple-graph-grammars [22].

Continuing the above example, the subsequent paragraphs concentrate on the execution semantics for each of the operators given in Fig. 3, by using ATL-like code templates. At compile-time, each operator is derived into a fragment of ATL-code, only. A weaving in a certain language, though, stands for a

[1] For simplicity reasons code snippets use simplified ATL syntax.

complete ATL transformation. The generator, therefore, needs to integrate all these fragments into a complete ATL transformation as shown in Fig. 3.

Fig. 4 depicts pseudo-template code to show how semantics of operators can be specified. The template code consists of target code (ATL) in plain text, and template code in angle brackets which is bound at compile-time against LHS and RHS model elements. Square brackets contain control-flow instructions for the generator. In the template body of the parenting *FullEquivClass* operator for instance, templates of children operators are invoked.

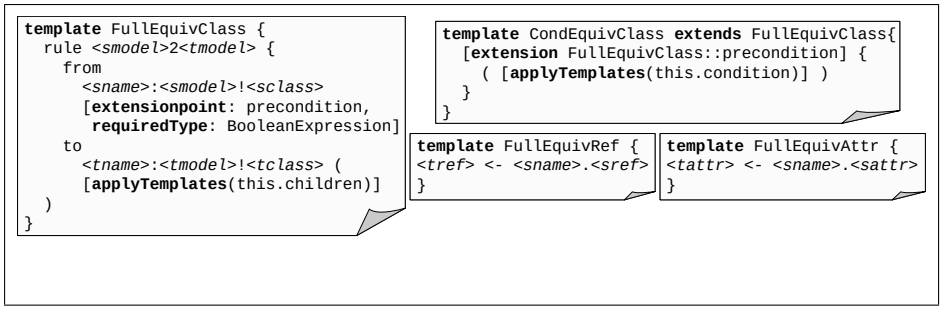


Figure 4: Template code for integration operators.

To enable the extension of existing operators, a plugin-mechanism can be used. Thereby, templates can offer extension-points, into which templates of more specialized operators can plug-in their contributions. In Fig. 4, the *FullEquivClass* template declares an extension point that requires the contribution of a boolean expression. An example for an extension is given by the template of the *CondEquivClass* operator, which itself invokes a template that returns a boolean expression bound to the operator's context. Through this inheritance-based reuse, a *CondEquivClass* operator can inherit all of *FullEquivClass*' behavior and additionally denote that a model element should be transformed if a certain condition holds, only.

5 Related Work

In this paper we have laid out an approach stretching across various abstraction layers, from global model management to local MTLs. As shown in Table 3, existing work typically focuses on certain abstraction levels, but, in our opinion, have not established a common understanding of how bottom-up approaches can be utilized for the implementation of top-down approaches in a scalable way. Furthermore, we compare related works on basis of certain key characteristics of our approach, like the employment of DSILs, OO-style model management scripts, the extensibility of operators and the explicit use of declarative integration specifications.

Related Work	Key Characteristics				Abstraction Levels				
	DSIL	OO	Extensible	Declarative	Glob. Comp.	Glob.	Intermed. Comp.	Loc.	Loc.
MMgmt.	-	-	-	+	+	+	-	-	-
MOMENT	-	-	~	+	-	+	-	-	+
GGT	+	-	-	+	-	+	+	+	~
AMW	+	-	+	+	-	-	+	~	-
EOL	+	-	+	~	-	-	+	+	+
ATL	-	-	-	~	-	-	-	-	+

Table 3: Comparison of related work.

Model management as proposed by Bernstein et al. aims at applying operators on the model level [3] [12]. In [4] a language-independent semantics is established to guide the implementation of model management operators. Although our work embraces the ideas of model management operators, e.g., by categorizing transformations, we also extend the notion of model management scripts with OO-mechanisms and explicitly focus on providing for scalable implementations through DSILs.

MOMENT [10] realizes model management operators by defining their semantics in QVT relations [23] that are mapped onto the algebraic specification language Maude, which, through term rewriting, executes the defined transformations. Although we focus on supporting the implementation of model management operators, the justified intention behind MOMENT to study formal properties of transformations could complement our approach in the future. However, our approach could potentially do this on the more abstract level of basically language independent integration operators and DSILs, as opposed to MOMENT, where Maude doubles as an execution environment as well as a testbed for proving formal properties.

The *Glue Generator Tool* (GGT) [8] aims at the reuse of existing MDA applications by specifying composition relationships between platform-independent models (PIMs), of which glue code for the integration of platform-specific models (PSMs) can be derived. Although rules similar to our integration operators are offered, our approach seems to be more flexible as we allow to extend the semantics of integration operators. Furthermore, the integration scenario described in GGT could be realized as a model management script carrying out the necessary transformations, which could allow for better modularity and maintainability of the overall approach.

The *Atlas Model Weaver* (AMW) [7] is a generic, extensible tool that aims at supporting modelers to establish semantic links between elements of arbitrary models or metamodels. The links are referred to as weavings and are formalized in a weaving metamodel, which can be extended to denote link types with special semantics. This extension mechanism is the basis for defining the syntax of integration operators and DSILs in our approach. Created weavings can then be subject to further processing like derivation of MTL code.

The *Epsilon Object Language* (EOL) is a language for managing models of arbitrary metamodels [21]. It can either be used as a standalone language for model navigation and comparison, or also as an infrastructure on which task-specific languages such as the *Epsilon Merging Language* (EML) or the *Epsilon Comparison Language* (ECL) can be built. Similarly, the *Atlas Transformation*

Language (ATL) [1] is a hybrid (imperative/declarative) MTL based on the Eclipse Modeling Framework. In our opinion, both efforts present themselves as possible execution environments for our approach. Especially the definition of execution semantics for DSILs falling into categories like *Check* or *Fusion* could be conveniently accomplished relying on the expressiveness of languages like ECL or EML.

6 Conclusion and Future Work

In this paper we have proposed a conceptual approach which allows to define declarative model integration languages to implement model management operators, and to compose these into model management scripts. The distinction between local and global transformations fosters reuse of existing integration operators, and allows for sound composition of transformation functions. We have given a description of transformation categories and exemplified the composition of transformations into model management scripts. According to the understanding of transformations defining the *integration behavior* of metamodels, these scripts rely on an OO mechanism to invoke transformations which are dynamically bound depending on a metamodel's type. Furthermore, we discussed the syntax and the semantics of an example integration language and described a way to extend integration operators.

We think of the approach described in this paper as a step towards the realization of future transformation systems which operate on the global model level, as opposed to the local model-element level, only. To raise the level of abstraction, domain specific languages in the form of declarative integration specifications play a key part in our approach. These are built on existing general-purpose transformation languages and are basically technology neutral. We have experimented with the implementation of various weaving languages which consist of operators that form the language kernels for the proposed transformation categories. Current work deals with building a technical framework based on existing model engineering infrastructure supporting our approach and a generically reusable toolset for various transformation categories.

In the context of ModelCVS, besides the integration of modeling tools, a crucial issue is the support for language evolution through metamodel modification. Future work will investigate to what extent such metamodel extensions can have characteristics analogous to traditional OO sub-classing, which would allow transformations to be inherited towards extended versions of metamodels.

Acknowledgement

We thank Elisabeth Kapsammer, Wieland Schwinger and Manuel Wimmer for their comments.This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.

References

[1] ATL Homepage, http://www.eclipse.org/gmt/atl/, 2006.

[2] Batory, D., *Multilevel models in model-driven engineering, product lines, and metaprogramming.* IBM Systems Journal, VOL 45, NO 3, 2006.

[3] Bernstein, P.A., *Applying Model Management to Classical Meta Data Problems.* In Proceedings of the Conference on Innovative Data Systems Research (CIDR), Asilomar, California, January 2003.

[4] Bernstein, P.A., A.Y. Halevy, S. Melnik, and E. Rahm, *A Semantics for Model Management Operators.* Microsoft Technical Report, June 2004.

[5] Bernstein, P.A., S. Melnik, and E. Rahm, *Rondo: A Programming Platform for Generic Model Management.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2003.

[6] Bézivin et al., *Combining Preoccupations with Models.* 1st Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD at the 19th ECOOP, July 2005.

[7] Bézivin, J., E. Breton, M. Didonet Del Fabro, G. Gueltas, and F. Jouault, *AMW: A Generic Model Weaver.* In Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles, Paris, France, 2005.

[8] Bézivin, J., F. Jouault, D. Kolovos, I. Kurtev, and R.F. Paige, *A Canonical Scheme for Model Composition.* A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 346–360, 2006.

[9] Bézivin, J., F. Jouault, P. Rosenthal, and P. Valduriez, *Modeling in the Large and Modeling in the Small.* LNCS, No. 3599, edited by Uwe Aßmann, Mehmet Aksit, Arend Rensink. Springer-Verlag GmbH, pp. 33–46, 2005.

[10] Boronat, A., J.Á. Carsí, and I. Ramos, *Algebraic Specification of a Model Transformation Engine.* European Joint Conferences on Theory and Practice of Softwaere (ETAPS06), Vienna, March 2006.

[11] Bouzitouna, S., M.P. Gervais, and X. Blanc, *Models Reuse in MDA.* In Proceedings of the International Conference on Software Engineering Research and Practice (SERP05), Las Vegas, USA, June 2005.

[12] Brunet et al., *A Manifesto for Model Merging.* In Proceedings of the 1st International Workshop on Global Integrated Model Management (GaMMa2006), Shanghai, May 2006.

[13] DeRemer, F., and H. Kron, *Programming-in-the-Large Versus Programming-in-the-Small.* IEEE Trans. on Soft. Eng. 2(2), 1976.

[14] Engel, K.-D., D.S. Kolovos, and R.F. Paige, *Using a Model Merging Language for Reconciling Model Versions.* A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 143–157, 2006.

[15] Gurevich, Y., P. Kutter, M. Odersky, and L. Thiele (eds.), *Abstract State Machines: Theory and Applications.* LNCS VOL 1912, Springer-Verlag, 2000.

[16] Jouault, F., *Loosely Coupled Traceability for ATL.* In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany, 2005.

[17] Kappel et al., *Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages.* In Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML), Genova, Italy, October 2006.

[18] Kappel et al., *On Models and Ontologies - A Semantic Infrastructure Supporting Model Integration.* In Proceedings of Modellierung, Innsbruck, Tirol, Austria, March 2006.

[19] Kappel et al., *Towards A Semantic Infrastructure Supporting Model-based Tool Integration.* In Proc. of the 1st Int. Workshop on Global integrated Model Management (GaMMa2006), Shanghai, May 2006.

[20] Kapsammer, E., T. Reiter, W. Retschitzegger, and W. Schwinger, *Model Integration Through Mega Operations.* In Proc. of the Int. Workshop on Model-driven Web Engineering (MDWE), Sydney, July 2005.

[21] Kolovos, D.S., R.F. Paige, and F.A.C. Polack, *The Epsilon Object Language (EOL).* A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 128–142, 2006.

[22] Königs, A., and A. Schürr *Specification of Graph Translators with Triple Graph Grammars.* In Proc. of Graph-Theoretic Concepts in Computer Science, 20th Int. Workshop, Herrsching, Germany, 1994.

[23] Object Management Group (OMG), *MOF QVT Final Adopted Specification.* November 2005.

[24] Reiter, T., W. Retschitzegger, W. Schwinger, and M. Stumptner, *A Generator Framework for Domain-Specific Model Transformation Languages.* In Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS), Paphos, Cyprus, May 2006.

[25] Stumptner, M., M. Schrefl, and G. Grossmann, *On the Road to Behavior-Based Integration.* In Proceedings of Conceptual Modelling, First Asia-Pacific Conference on Conceptual Modelling (APCCM2004), Dunedin, New Zealand, January 2004.

[26] Wiederhold, G., P. Wegner, and S. Ceri, *Toward megaprogramming.* CACM, Volume 35, Issue 11, pp. 89–99, November 1992.

Block Diagrams as a Syntactic Extension to Haskell

Ben Denckla
Denckla Consulting, 1607 S. Holt Ave.,
Los Angeles, CA 90035, USA
bdenckla@alum.mit.edu

Pieter J. Mosterman
The MathWorks, Inc., 3 Apple Hill Dr.,
Natick, MA 01760, USA
pieter.mosterman@mathworks.com

Abstract. Often, the semantics of languages are defined by the products that support their usage. The semantics are then determined by the source code of those products, which often is a general-purpose programming language. This may lead to complications in defining a clean semantics, for example because imperative notions slip into a declarative language. It is illustrated how block diagrams can be translated into Haskell to define the semantics of a graphical language in terms of a textual programming language. This also allows the use of block diagrams as a syntactic extension to Haskell and the use of Haskell as an action language in block diagrams. Imperative notions can then be included from the declarative perspective of Haskell, which is more constrained and less prone to resulting in complicated semantics of interaction and combination of the imperative and declarative.

1 Introduction

The increasing application of computational power in engineered systems such as automobiles, consumer electronics, and aircraft has resulted in a staggering complexity that has proven difficult to negotiate with conventional design approaches. For example, high-end automobiles may now employ up to 80 microprocessors that largely interact with each other through the automobile networks such as the controller area network, CAN [1]. The discrete nature of the software that is running on these microprocessors abandons the notions of continuity that are inherent in the more traditional physics design problems [20] and this has put forward the need for different design approaches.

To successfully tackle the computational complexity of modern engineered systems, Model-Based Design [3] is increasingly being adopted, which addresses the software complexity at a computational model level. This allows circumventing the practice of designing functionality directly at the level of computation, such as in assembly or programming languages. The power of those languages often leads to complexity in a design that is difficult to comprehend, and, therefore, results in errors. To curtail this problem, typically programming styles are

mandated and code structure of too high complexity (e.g., cyclomatic complexity) is disallowed.

Though styles have been successful to mitigate the error-prone nature of design at a programming language level, it has not addressed the design problem satisfactorily. For example, the design of the F/A-22 has still not been completed in spite of significant budget increases, which has large been caused by software producibility problems [23].

Rather than designing the structure of computation in a programming language, Model-Based Design allows the use of high-level and domain-specific languages. These languages can be tailored to capture the semantic notions of the domain in which the design problem needs to be solved. This allows the design engineers to work with a language that is intuitive and close to their understanding of the problem, and, therefore, is more efficient and less error-prone.

There are a number of essential aspects to this approach:

- The design of tailored languages needs to be an efficient process itself.
- The evolution of a given language needs to be supported.
- The domain-specific language needs to be automatically transformed into a structure of computation.

The field of Computer Automated Multiparadigm Modeling (CAMPaM) [19] aims to address these aspects by establishing a framework to reason about models of systems at multiple levels of abstraction, transforming between models in different languages, and providing and evolving modeling languages. This paper concentrates on the language aspect and shows how a graphical model can be designed as syntactic extension to a textual model. It furthermore touches upon the differences between imperative and declarative semantics that is orthogonal to the language modality and it discusses the interaction between the two.

Section 2 provides a brief introduction to CAMPaM to establish a common vocabulary and show where this work fits into the overall framework. Section 3 discusses the graphical and textual modalities and their characteristics. Section 4 concentrates on block diagrams specifically and presents the block diagram syntactic extensions to Haskell [15], a functional and declarative language. Section 5 presents the conclusions of this work.

2 Computer Automated Multiparadigm Modeling

The basic CAMPaM aspects are introduced and important notions that it entails are defined.

2.1 Aspects of CAMPaM

Previous work [19] has established CAMPaM as the field that provides a framework and computational methods to relate and combine models. This requires handling the different levels of abstraction that are being used for the models as well as the different formalisms that are being employed. Note that the levels of abstraction and different formalisms are orthogonal, although often a different formalism is employed for a different level of abstraction. This makes formalisms a first class element of study, and establishes the modeling of formalisms as an essential activity, where a formalism can be thought of as a having a syntax and semantics [13].

All in all, this puts forward the following aspects of CAMPaM:

- multiple levels of abstraction,
- multiple formalisms,
- formalism modeling, which includes
  - modeling the syntax, and
  - modeling the semantics.

This paper concentrates on formalisms, in particular on combining and relating formalisms and the elements of a formalism.

2.2 Definitions

To anchor the work presented in this paper, a set of definitions are given that are derived from discussions during a number of workshops on Computer Automated Multiparadigm Modeling[1] [2]. A detailed description of this framework and a set of agreed upon definitions, which may differ from the following, is forthcoming.

The Ogden/Richards semiotic triangle[2] can be thought of as establishing a relationship between concrete syntax, abstract syntax, and semantics. The concrete syntax is the actual referent. The abstract syntax is a symbol representing the referent. The semantics is the thought associated with the referent. This work restricts the referent to be a sentence.

Definition 1 (Sentence)**.** A presentation of information.

A sentence in a given language is presented in its concrete syntax.

Definition 2 (Concrete Syntax)**.** The presentation of a sentence is in the concrete syntax of a language.

A sentence can be represented as an element of an abstract syntax.

Definition 3 (Abstract Syntax)**.** The abstract syntax contains representations of sentences.

[1] http://moncs.cs.mcgill.ca/people/mosterman/campam/
[2] http://en.wikipedia.org/wiki/Semiotic_triangle

In general, an abstract syntax consists of a set of symbols and their possible combinations. This then requires the abstract syntax to be comprised of entities, the symbols, and relations, the combinations of symbols.

A sentence relates to information by invoking a thought when it is interpreted. This thought is the semantics of the sentence.

Definition 4 (Semantics)**.** The semantics of a sentence is the thought associated with that sentence.

A sentence is an element of a set of sentences that is called a language.

Definition 5 (Language)**.** A language is a set of sentences.

The sentences in an abstract syntax relate to a set of semantics that is called the semantic domain of the language that represents the set of sentences.

Definition 6 (Semantic Domain)**.** The semantic domain of an abstract syntax is the set of thoughts associated with the elements in the abstract syntax.

The semantic domain then consists of the intuitive notions that can be represented by the elements of the abstract syntax.

To associate a thought with a sentence is to give it a meaning. This corresponds to a mapping of the sentence to a semantics.

Definition 7 (Meaning)**.** The meaning of a sentence in the abstract syntax is a projection into the semantic domain of the language.

The combination of an abstract syntax, its semantic domain, and a meaning for each of the elements in the abstract syntax is called a formalism.

Definition 8 (Formalism)**.** A formalism consists of an abstract syntax, a semantic domain, and a meaning.

A semantic domain can be shared completely or partially by many formalisms.

A sentence can be translated by changing its syntax.

Definition 9 (Translation)**.** The translation of a sentence is another sentence in another formalism.

An abstract syntax can be modeled by another or the same abstract syntax, where the model represents a set of sentences, or language. The language model is called a metamodel and its meaning is the abstract syntax of the language that is modeled.

Definition 10 (Metamodel)**.** The meaning of a sentence in a metamodel language is an abstract syntax.

This metamodel can be a grammar or an enumeration and may be generative.

The metamodel does not model a formalism, as it does not capture the meaning of the abstract syntax. Because the meaning is a relation between abstract syntax and thought, no explicit model of meaning can be provided. Instead, the meaning is modeled implicitly by a relation between two abstract syntaxes (that may be the same), an interpretation, where the range abstract syntax is more intuitive than the domain abstract syntax.

3 Textual and Graphical Languages

The concrete syntax of a language is the presentation of the abstract syntax. This presentation is often classified as either graphical or textual. At the abstract syntax level, there is no principled distinction between the two.

3.1 Textual Versus Graphical Languages

Whether textual or graphical syntax is preferred depends much on the application. Textual syntaxes of many written natural languages, programming languages such as FORTRAN and C, and modeling languages such as Modelica™ [11] have proven to be useful and successful. Similarly, graphical syntaxes of modeling languages such as bond graphs [17], Petri nets [4], and block diagrams [6] have been very successful in their respective usages as well.

At the core, the difference between graphical languages and textual languages is that the former often reference expressions by lines (directed or undirected) whereas the latter reference expressions by symbols. The use of lines allows direct display of referencing relations between expressions, which helps in quickly building an understanding of the model. The drawback of this explicitness is that it becomes costly in terms of visual clutter when many such references are present. Though the use of symbols for referencing prevents the graphical clutter, it may lead to clutter of the namespace with named references.

Considering graphical models that represent programs, one could say the value of graphical models is that they allow the graphical structure of a program to be expressed in a more direct form. Textual syntaxes have to represent the program (term graph) mostly as a tree, and further, have to represent that tree as a sequence of lexemes. They represent a graph mostly as a tree by adding some context-sensitive constraints to a context-free grammar. They represent a tree as a sequence through mechanisms such as parentheses and precedence. This is not to say that the more direct, graphical form of expression offered by graphical models is always preferable, on the contrary. Most textual programs can be seen as compact, elegant representations of what would be a very convoluted graph. On the other hand, there are times when it would be clearer to see the graph directly, and this is what graphical models offer.

It is also possible to imagine a language that has a single notion of reference with two different ways to view it: one by arrow and one by symbol. So, rather than arguing a purified approach, a mixture of these approaches is advocated and typically provided in industrially successful products. For example, the GoTo and From blocks that Simulink® [22] provides allow reference by symbol in addition to the graphical referencing inherent in block diagrams. Another example of this mixture of referencing is Subtext [7] and, in a more limited way, the many integrated development environments (IDE) that allow the user to, e.g., jump from a site where a symbol is used to the symbol definition.

In general, graphical languages are never full languages; they are always combined with some other language that defines what individual entities and relations mean. For example, the meaning of a block in a block diagram is

typically defined by a name inside the block (e.g. "H"), possibly combined with the use of a shape other than a rectangle for a block, (e.g., the use of a triangle instead of a rectangle, to indicate a gain, with the gain factor indicated by a numeral such as "-1" inside the block).

3.2 Related Work

Ellner and Taha [9, 10] provide a good introduction to the unneeded gap between block diagram and textual languages. They also provide a promising way to formally bridge this gap with a visual multi-stage calculus called PreVIEW. The work presented in this paper seeks to bridge this gap in a different but complementary way. It does not intend to provide a visual language equivalent to a textual one. Rather, a textual language (Haskell) is only extended with visual features. Additionally, the work presented in this paper does not address multi-stage programming and it concerns a full language because of the use of Haskell. This is in contrast to the use of a minimal calculus that is suitable for theoretical work, as presented by Ellner and Taha.

The Subtext language [7] is notable for the way it bridges the textual/graphical gap. Names (identifiers) are present in Subtext but are more like comments upon its fundamental concepts of sources and references connected by links, which are represented explicitly. The HOPS language is notable [16] for its implementation of a Haskell-like language using a term graph instead of a textual representation. The Vital language [12] is notable for its implementation of a Haskell environment in which data structures can be viewed and manipulated as diagrams.

The particular idea of implementing a block diagram language as syntactic extensions to Haskell is discussed and partially implemented by Reekie [21]. This is closely related to the approach of implementing a domain-specific language by embedding it in a general-purpose language [14]. However, in this case, the block diagram language needs new syntax, so it cannot, in the strictest sense, be called embedded. Implementing a new language by extending syntax or by embedding can be seen as part of Landin's 40-year-old program to avoid reinvention of general-purpose language capabilities when inventing a new language [18]. Of course this is not always possible and desirable, but the work presented in this paper intends to illustrate that it is in the case of block diagrams.

Again, the point made is that it is not imperative to make a choice: the approaches can be mixed by extending a textual language with block diagrams.

3.3 Semantics of Block Diagrams

Block diagrams have become very popular among control system engineers to support their design efforts. In particular, the support for computational simulation by block diagram based products such as Simulink has been an important enabler for this success. Furthermore, the constraints that are enforced by block diagrams address the specific needs of the control system discipline. The corresponding limitations in expressiveness, as compared to general pur-

pose programming languages, prevent the user from making mistakes that can happen in a general language of computation. For example, Simulink supports only a very stylized form of recursion which cannot cause infinite loops or stack overflows.

On the other hand, the semantics of block diagram languages are often informal, which suffices for many users who infer the semantics through trial and error or by example, and do not rely on a documented semantics.

Because a formal semantics is not required for most successful applications of a language and to achieve expedient and convenient implementation of interpreters, often a programming language such as C is used as the language for defining the semantics. For example, the semantics of Modelica is defined by the source code of its supporting products. It is evident that semantic discrepancies between products that support the same language are inevitable.

Furthermore, the freedom of expression that programming languages such as C allow can lead to semantic inconsistency when adding language features. For example, in a pure interpretation, block diagrams are a declarative language, but the use of an imperative language to define the semantics of new language elements has allowed introducing imperative notions in the form of the Merge block in Simulink. This block produces as output the last computed input, and, therefore, the outcome becomes dependent upon the execution order. In case of the Merge block, the use of imperative notions has lead to a very powerful language construct that greatly aids the modeling task. In other words: "One user's feature is another user's bug."

Another implication of the use of an imperative language to define the semantics of a declarative language is that it becomes difficult to integrate imperative notions with the declarative language in a consistent manner. In the implementation, there is no distinction between the two, and, therefore, maintaining a consistent separation and interaction becomes problematic.

History has shown, though, that such limitations constitute no impediment to the success of a language (natural languages included). However, in applications where categorical evaluations are required, a clear definition of semantics is often required. This paper aims to illustrate that such a definition can be achieved for block diagrams. In particular, the use of a declarative language such as Haskell to define the semantics prevents many of the issues discussed. Furthermore, because block diagrams rely on some other language to provide the symbolic environment in which blocks are defined, it is a short leap to make this environment a scope in Haskell.

4 Block Diagrams and Haskell

Building on previous work [5], BdHas is presented as Haskell [15] plus syntactic extensions for block diagrams. A block diagram can be used inside a Haskell expression, and a Haskell expression can be used inside a block. The interpretation of a BdHas program is the Haskell code that results from translating all

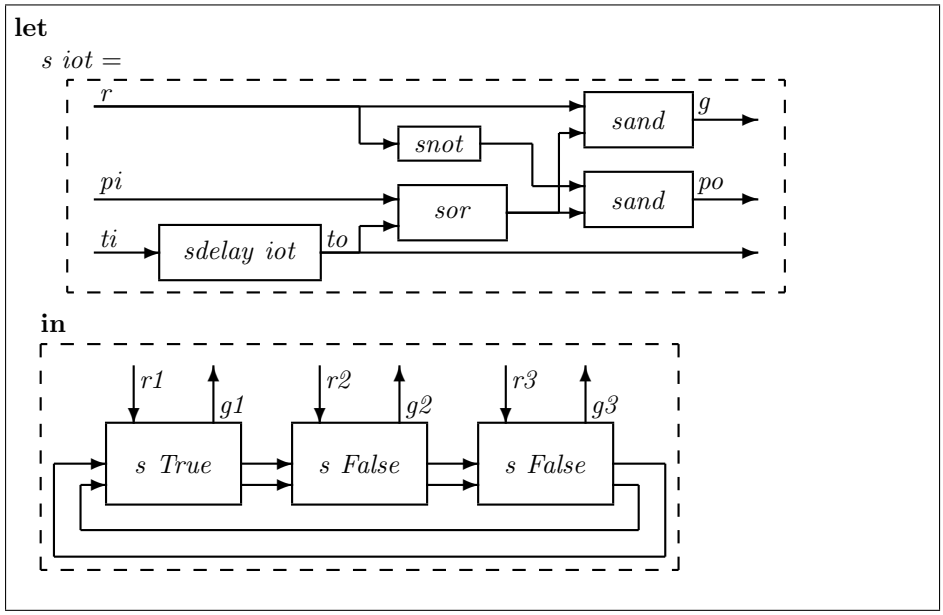


Figure 1: Token ring arbiter in BdHas

of its block diagrams to pure Haskell. Before describing this translation, an introduction to BdHas is provided by means of an example.

4.1 Introduction to BdHas

The example used to introduce BdHas is that of a simplified token ring bus [8]. In each clock cycle, the stations on the bus "decide amongst themselves" which station (if any) will use the bus. The decision is made in the following manner:

- A station has permission to use the bus if it owns the token or has been passed permission to use the bus.
- A station with permission to use the bus grants itself the bus if it has requested the bus.
- Otherwise it passes permission to use the bus to the clockwise-next station.
- At the end of each clock cycle, token ownership is transferred to the clockwise-next station.

Figure 1 shows the arbiter as a hierarchical block diagram in BdHas. The functions *sdelay*, *sand*, *sor*, and *snot* are assumed to be defined elsewhere to be a unit delay and stream versions of the boolean operators "and," "or," and "not," respectively. The symbol r is short for "bus requested," g for "bus granted," pi/po for "permission in/out," ti/to for "token ownership in/out," and *iot* for "initially owns token."

The symbols labeling arrow tails (r,g,pi, etc.) are comments to aid understanding; they have no semantic significance as they would in a textual program.

To execute this BdHas program, it is passed to a program that translates the syntactic block diagram extensions into pure Haskell, yielding the code in Fig. 2. Note that some automatically generated identifiers have been renamed to make them more meaningful.

Although hand-written code may be quite different, the automatically generated code is a reasonable implementation and thus conveys a notion of how advantageous the use of a block diagram can be over textual code.

```
let
  s iot =
      λ(r, pi, ti) →
        let
          g = sand (r, o)
          po = sand (n, o)
          to = sdelay iot ti
          o = sor (pi, to)
          n = snot r
          in
          (g, po, to)
  in
  λ(r1, r2, r3) →
    let
      (g1, po1, to1) = s True (r1, po3, to3)
      (g2, po2, to2) = s False (r2, po1, to1)
      (g3, po3, to3) = s False (r3, po2, to2)
      in
      (g1, g2, g3)
```

Figure 2: Token ring arbiter translated from BdHas to Haskell

4.2 Translating Block Diagrams to Haskell

The translation from BdHas to Haskell proceeds roughly as follows. Give a unique identifier (ID) to each arrow tail junction. An arrow tail junction (ATJ) is a point where one or more arrow tails coincide. Translate each block to a declaration of the form $y = f\ u$ where

- y is a tuple of the IDs of the ATJs on the block,
- u is a (possibly empty) tuple of the IDs of the ATJs for each arrow whose head is on the block, and
- f is the translation of the BdHas expression inside the block.

Translate the diagram to

$\lambda(i, i, ...) \rightarrow$
let
$(i, i, ...) = e\ (i, i, ...)$
$(i, i, ...) = e\ (i, i, ...)$
...
in
$(i, i, ...)$

Figure 3: General form of block diagrams translated to Haskell

- a λ expression binding a (possibly empty) tuple of the IDs of the ATJs that are not on any block, where the body of this λ expression is
  - a **let** expression containing the (possibly empty) list of declarations from the block translations, where the body of this **let** expression is
    - a tuple of the IDs of the ATJs for each arrow whose head is not on any block.

So, in general, block diagrams translate to expressions of the form shown in Fig. 3, where i is a meta-variable ranging over the IDs of the ATJs and e is a meta-variable ranging over the expressions that result from translating the contents of the blocks.

4.3 Integrating Imperative Notions

The desire to integrate imperative notions in block diagrams has been discussed in Section 3.3. In Haskell, it is straightforward to create a pseudo-imperative domain-specific embedded language in which integers can be added and an "instruction count" of the tally of additions at a point in the instruction sequence can be retrieved.

For example, with a few lines of helper code not shown, the following function

addSeq = **do**
  $x \leftarrow$ *add* 1 2
  $c \leftarrow$ *get*
  $y \leftarrow$ *add* x 3
  $z \leftarrow$ *add* y c
  return z

returns 7. But, if the line $c \leftarrow$ *get* is moved to one line later, i.e.,

addSeq = **do**
  $x \leftarrow$ *add* 1 2
  $y \leftarrow$ *add* x 3
  $c \leftarrow$ *get*
  $z \leftarrow$ *add* y c
  return z

it will return 8.

5 Conclusions

Domain specific languages help efficient and effective design of systems with a significant computational component. A classification into textual and graphical languages can be made. Whereas textual languages often lack a graphical component, graphical languages typically include textual extensions.

The definition of semantics for a language has been discussed as a syntactic translation. In many languages, the semantics are defined by the source code of a product that supports each language. This often leads to complications because of the implementation freedom that general-purpose programming languages such as C provide.

This paper discussed the use of Haskell to provide semantics for block diagrams instead. The declarative nature of Haskell renders it well-suited for this purpose. Furthermore, this provided the basis for including block diagrams into Haskell as a syntactic extension. The translation of block diagrams into Haskell results in a uniform pure Haskell representation. Additionally, this facilitates the use of a full programming language as an action language for block diagrams.

This paper has further briefly illustrated how imperative notions may be included in Haskell. Rather than using imperative programming languages to define declarative semantics, this may result in more rigorous and consistent definition of mixed imperative/declarative languages.

A working implementation based on a textual input of the block diagrams was presented while a visual editor is not yet available.

6 Acknowledgment

The authors wish to acknowledge the other attendees of the 2004 through 2006 editions of the *International Workshop on Computer Automated Multi-Paradigm Modeling*:[3] Jean-Sébastien Bolduc, Peter Bunus, Gary Godding, David Hill, Stephen Neuendorfer, Hans Vangheluwe, Mamadou Traore, Thomas Kühne, Hessam Sarjoughian, Vasco Miguel Moreira do Amaral, Adam Cataldo, Jerome Delatour, Jean-Marie Favre, Holger Giese, Anneke Kleppe, Juan de Lara, Tihamér Levendovszky, Jie Liu, Alexandre Muzy, and Ernesto Posse for their help in developing the CAMPaM framework.

References

[1] CAN specification. Technical Report, 1991. Robert Bosch GmbH.

[2] CAMPaM '06 Attendees. CAMPaM 2006 workshop position statements. Technical Report SOCS-TR-2006.2, McGill School of Computer Science, 2006.

[3] http://moncs.cs.mcgill.ca/people/mosterman/campam/

[3] Paul Barnard. Graphical techniques for aircraft dynamic model development. In *American Institute of Aeronautics and Astronautics (AIAA) Modeling and Simulation Technologies Conference and Exhibit*. Providence, Rhode Island, August 2004. CD-ROM, paper number AIAA-2004-4808.

[4] René David and Hassane Alla. *Petri Nets & Grafcet*. Prentice Hall Inc., Englewood Cliffs, NJ, 1992. ISBN 0-13-327537-X.

[5] Ben Denckla, Pieter J. Mosterman, and Hans Vangheluwe. Towards an executable denotational semantics for causal block diagrams. In *Proceedings of The 5th OOPSLA Workshop on Domain-Specific Modeling*, San Diego, CA, October 2005. ISBN 951-39-2202-2.

[6] Richard C. Dorf. *Modern Control Systems*. Addison Wesley Publishing Co., Reading, MA, 1987.

[7] Jonathan Edwards. Subtext: uncovering the simplicity of programming. In *Proceedings of OOPSLA '05*, pages 505–518. ACM Press, 2005.

[8] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.

[9] Stephan Ellner. PreVIEW: An untyped graphical calculus for resource-aware programming. Master's thesis, Rice U., 2004.

[10] Stephan Ellner and Walid Taha. The semantics of graphical languages. In *Informal Proceedings of the Workshop on Designing Correct Circuits*, 2006.

[11] Hilding Elmqvist *et al.* Modelica^{TM}–A unified object-oriented langauge for physical systems modeling: Language specification, December 1999. version 1.3, http://www.modelica.org/.

[12] Keith Hanna. http://www.cs.kent.ac.uk/projects/vital/index.html.

[13] David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff. Technical Report MCS00-16, The Weizmann Institute of Science, 2000.

[14] Paul Hudak. Modular domain specific languages and tools. In *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[15] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge U. Press, April 2003.

[16] Wolfram Kahl. The term graph programming system HOPS. In *Tool Support for System Specification, Development and Verification*, pages 136–149, March 1999.

[17] D.C. Karnopp, D.L. Margolis, and R.C. Rosenberg. *Systems Dynamics: A Unified Approach*. John Wiley and Sons, New York, 2 edition, 1990.

[18] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[19] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 80(9):433–450, September 2004.

[20] Henry M. Paynter. *Analysis and Design of Engineering Systems*. The M.I.T. Press, Cambridge, Massachusetts, 1961.

[21] H. J. Reekie. *Realtime Signal Processing: Dataflow, Visual and Functional Programming*. PhD thesis, U. of Technology at Sydney, Australia, 1995.

[22] Simulink®. *Using Simulink®*. The MathWorks, Inc., Natick, MA, March 2006.

[23] Michael Sullivan. TACTICAL AIRCRAFT–F/A-22 and JSF acquisition plans and implications for tactical aircraft modernization. Technical Report GAO-05-519T, United States Government Accountability Office, April 2005.

An Integration Concept for Complex Modelling Techniques

Benjamin Braatz
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
bbraatz@cs.tu-berlin.de

Abstract. In this paper a concept for the integration of complex modelling techniques like e. g. UML is proposed. The integration is done by translating complex models consisting of parts following different modelling paradigms into a common low-level language, which is designed to be minimalistic enough to serve as a source for code generation and verification. On the other hand the low-level language should be expressive enough to allow the integration of the most common structural, behavioural, and constraint modelling languages. As an example for a complex modelling technique a derivation of the UML, which focuses on a small subset of the UML diagrams, but also adds some additional techniques, is considered. Moreover, a low-level language for object-oriented modelling techniques is sketched.

Keywords: UML, Semantic Integration, Code Generation

1 Introduction

Contemporary modelling techniques follow a large variety of different paradigms. For example, the Unified Modeling Language (UML, see [7]) contains state machines, activity diagrams, sequence and collaboration diagrams, and the Object Constraint Language (OCL, see [8]) to describe the behaviour of a modelled system. Other techniques like structured flowcharts as introduced by Nassi and Shneiderman (see [6]), graph transformation systems (see e. g. [4]), different kinds of process algebras and Petri nets, and temporal or modal logic formalisms are also in common use.

It would be very desirable to be able to use multiple techniques from this wealth of possibilities in one project, in order to describe the various aspects of the system with the technique, which fits best. This approach can, however, only develop its full potential, if the interconnections between the used techniques are made explicit. Ideally all applied techniques should be semantically integrated, i. e. interpreted in a common semantic domain.

This paper proposes an abstract concept introduced in Sect. 2 for integrating complex, multi-paradigm modelling techniques by translating complex models into a low-level language, which is designed to facilitate the defintion of a formal semantics, the generation of code from models, and the formal verification of models.

In Sect. 3 a complex modelling technique based on the UML is introduced, which is then translated exemplarily into an object-oriented low-level language sketched in Sect. 4. Finally, in Sect. 5 the approach is summarised, related work is discussed and some ideas for future work are given.

2 Abstract Integration Concept

The concept proposed in this paper assumes a complex modelling technique, given by a class **Mod** of models containing parts, which are described following multiple paradigms. These models are translated into a low-level language, given by a class **L3** of low-level models, leading to a function Int: **Mod** → **L3**. The low-level language should contain the essential information included in the complex models, but encode it in a paradigm-independent way. Each low-level model $L \in$ **L3** can be decomposed into its constructive part Beh(L) and its descriptive part Prop(L), which share a common structural part Str(L). (Cf. Fig. 1.)

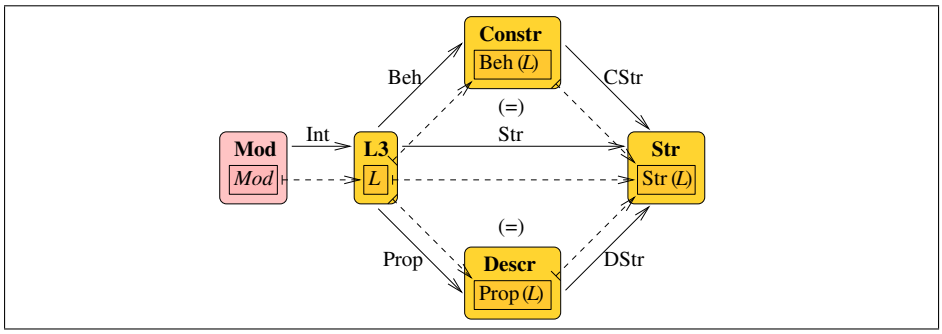


Figure 1: Relations between complex and low-level models

The rationale behind this decomposition is the strict seperation of the modelled behaviour and the properties, which have to be fulfilled by the behaviour. This seperation aids in providing tools for the low-level language. The constructive behaviour models can be used as a basis for code generation and as the axioms or models in verification, while the descriptive property models are suited to serve as a foundation for tests and as the target or specification in verification. The common structure models are needed to facilitate the connection between both parts of an **L3** model.

Note, that the seperation of concerns is established only in the low-level language. The complex modelling technique may, and most often will, combine constructive, descriptive, and structural aspects in the same diagrams. For example, a UML class diagram contains mostly structural information, but may also exhibit constraints like the multiplicities of attributes and associations or OCL pre-post-conditions and invariants, which are to be translated into the descriptive part of the corresponding low-level language model. It may even contain some OCL body constraints for query operations, which do not alter the

state of the system. Such constraints already specify the complete behaviour of the operation in question and may therefore be translated into the constructive part of the **L3** model.

A formal semantics for the low-level language will be provided by assigning a class of formal system representations **Sys**(*Str*) to each structural model *Str*, a subclass $\mathrm{Sem}_{Str}(Prop)$ of all systems satisfying the specified properties to each descriptive model *Prop*, and a single constructed system $\mathrm{Build}_{Str}(Beh)$ to each constructive model *Beh*. (Cf. Fig. 2.)

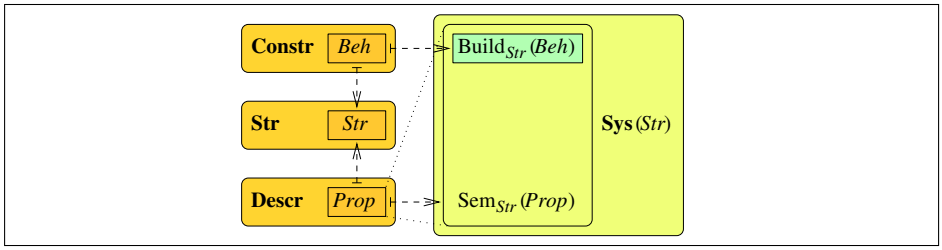


Figure 2: Semantics of low-level models

The exact definition of the semantic domain and functions is out of the scope of this paper. A draft version of such a semantics can be found in [3], which is based on transformation systems (see [5]) and adhesive high-level replacement systems (see [4]).

The formal semantics provides for the definition of consistency of a model by requiring that the behavioural part of the model satisfies the properties stated in the descriptive part, i. e. a low-level language model $L \in$ **L3** is consistent iff

$$\mathrm{Build}_{\mathrm{Str}(L)}(\mathrm{Beh}(L)) \in \mathrm{Sem}_{\mathrm{Str}(L)}(\mathrm{Prop}(L)) \ .$$

By composition with the integration function Int the formal semantics and the notion of consistency are also applicable to the complex modelling technique **Mod**.

The advantages of this approach lie in the decoupling of the complex modelling technique from the low-level language, for which the formal semantics is defined, and for which tools for generation and verification can be written. On the one hand, the semantics and tools do not have to deal with all the subtleties and "syntactic sugar" of the complex modelling technique, but can be based on the more minimalistic low-level language. On the other hand, the complex modelling technique can be enhanced by additional techniques rather easily, because only the class **Mod** and the integration function Int have to be adopted. Semantics and tools for extensions of the complex technique are then obtained automatically.

In the next section the idea of a complex integrated modelling technique will be illustrated by a small multi-paradigm modelling technique derived from the UML, while in Sect. 4 an object-oriented low-level language will be sketched.

3 CUML – The Complex Modelling Technique

As complex modelling technique we consider a derivation of the UML, which we will call CUML (Compact, Comprehensive, and Constructive UML). It uses only some of the features of UML, namely class diagrams, activity diagrams, and OCL constraints, but enhances them with transformation rules and structured flowcharts (also known as Nassi-Shneiderman diagrams, see [6]). These extensions were already proposed in [2] to yield a constructive, object-oriented modelling technique.

According to the intention of this paper, CUML is designed to allow code generation and formal verification. Therefore, we require stricter modelling than the original UML, which allows to leave a lot of features unspecified and describe requirements, actions, and other model properties in natural language.

We will describe the ideas along a small example of a shopping cart application, whose class diagram is shown in Fig. 3. The shopping cart itself is modelled by the Cart class containing Item instances, which in turn reference the corresponding Product instance. Instances of the class Catalog are used to administrate the Product instances.

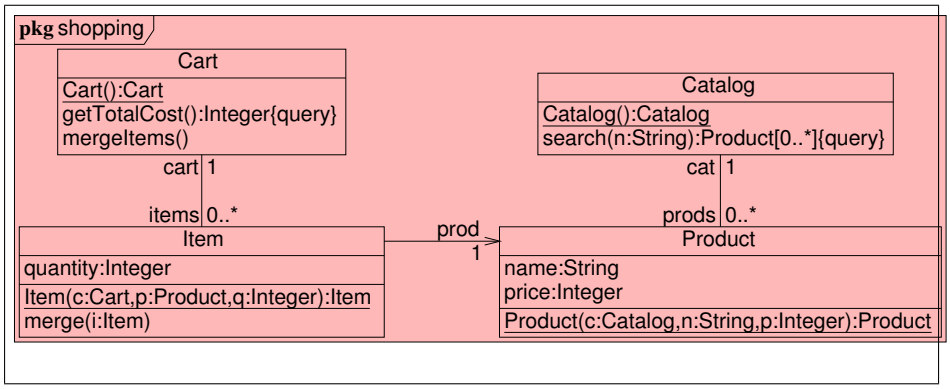


Figure 3: Class diagram of the example

Besides the usual features of class diagrams (declaring signatures of classes with properties and operations, as well as associations with multiplicities and navigability), we also use the possibility to specify if an operation is static for the constructors of the classes by underlining them and the possibility to specify if an operation is a query, i e. if it changes the state of the system.

These queries are on the one hand an operation for calculating the total cost of the items in a shopping cart, on the other hand a search operation on the products in a catalog. They are specified in Fig. 4(a) and 4(b) by OCL body constraints, which are an adequate choice because of the freeness from side effects in OCL. The total cost is calculated by iterating over the items in a shopping cart and adding the number mulitplied by the price of a single product, while searching is realised by the select operation predefined in OCL.

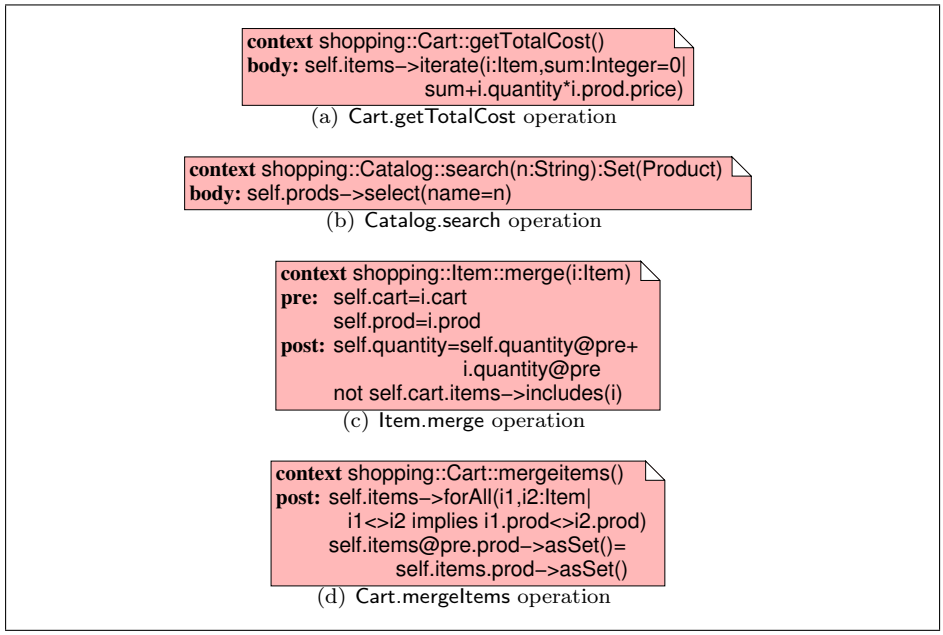


Figure 4: OCL constraints for the example

In contrast to the body constraints, the OCL pre-post conditions in Fig. 4(c) and 4(d) do not model the corresponding operations completely but merely specify some requirements. These operations are intended to merge items in the shopping cart, which reference the same product.

In order to model operations for local changes of the system, CUML uses a transformation rule notation as shown in Fig. 5. The advantage of this notation over activity diagrams and similar techniques is the declarative nature of transformation rules, which make the effects of operations on the object configuration readily visible by showing the relevant part of the system before the operation on the left-hand side and after the operation on the right-hand side.

The constructor for products in Fig. 5(a) creates a new product with the given name and price and adds the result to the given catalog, while the constructor for items in Fig. 5(b) creates a new item in the given shopping cart associated to the given product with the given quantity. The third rule in Fig. 5(c) models the behaviour of the operation for merging items in a shopping cart. The left-hand side of this rule shows that the operation is only applicable if the cart and the product of the item on which the operation is called and the parameter item are identical. The right-hand side then models the effect of the operation, where the parameter item is deleted and the quantities of self and parameter item are accumulated in self.

While OCL constraints follow a functional side-effect free paradigm suitable for modelling queries and transformation rules realise a declarative approach adequate for local changes of the system, the third behavioural technique we want to use in CUML is an imperative one, which can be used to model algo-

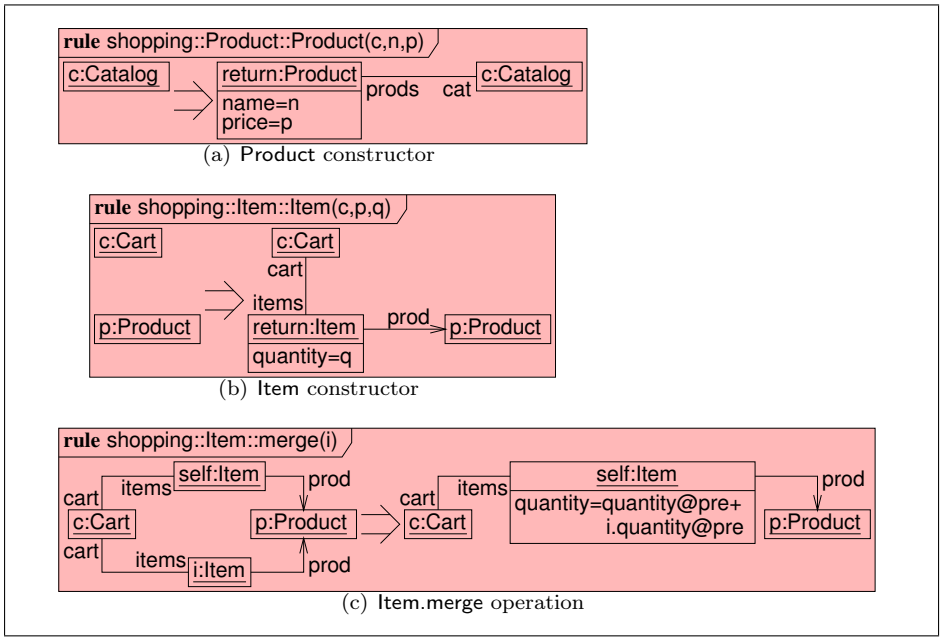


Figure 5: Transformation rules for the example

rithmic operations. In Fig. 6, we see a structured flowchart for the operation merging all items with identical products in a shopping cart.

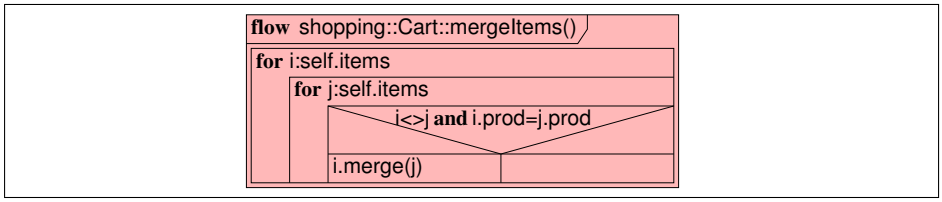


Figure 6: Flowchart for Cart.mergeItems operation

These structured flowcharts are a derivation of the flowcharts of Nassi and Shneiderman in [6]. In the example we can see that one of the advantages over graph-like techniques like activity diagrams and state machines is the visibility of the algorithmic structure with two nested iterations and a decision. Such algorithmic details are complicated to model adequately in graph-like modelling techniques.

However, we also want to integrate activity diagrams into CUML, because they are an adequate modelling technique for operations, which are less algorithmic, but rather workflow-like, though we do not have an example for such an operation in our small example.

In the next section we sketch, how the different techniques presented in this section may be integrated into a common object-oriented low-level language.

4 L3 – The Low-Level Language

As already stated in the abstract concept in Sect. 2, the low-level language will be subdivided into structural, descriptive, and constructive aspects. We will use a notation close to the UML notation for low-level models. In an implementation of this concept, low-level models would probably not be visualised at all, but rather only used in the backend, so that the developer only has to deal with (C)UML diagrams.

In Fig. 7, the structural part of the low-level model of the example is shown. It is still very similar to the class diagram, but abstracts from properties, which have to be fulfilled by the implementation rather than being ensured directly by the structure. Associations and properties are both translated into attributes, where the inverseness of the association ends will be required in the descriptive part. Likewise multiplicities are only considered in deciding if an attribute or parameter is a reference to a single object or a collection of objects, representing the multiplicities `0..1` and `0..*`, respectively. Other multiplicities will also be considered in the property model.

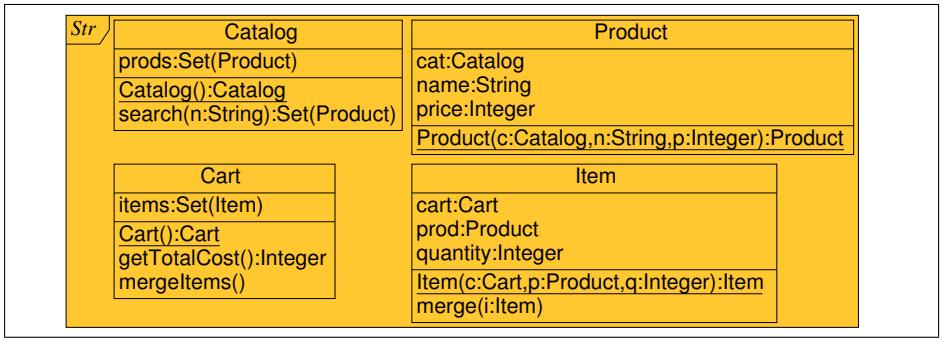


Figure 7: Structural low-level model of the example

In order to be able to represent all kinds of collections available in the UML, the low-level language should support sets, bags, sequences, and ordered sets. These collection types are retained in the low-level model rather than being flattened, because the code generation is likely to be able to translate them to structures provided by the underlying platform, e. g. the Java Collection Framework, directly.

The descriptive model in Fig. 8 contains translations of the association and multiplicity constraints from the class diagram and the OCL pre-post conditions, where all these are translated into first-order logic. A special keyword **query** is introduced to capture the property of an operation being a query.

Associations and multiplicities are invariants of object configurations. In order to be able to reuse them, abbreviations for these invariants are defined in the upper part of Fig. 8. Invariants are added to all pre and post conditions of all operations (except for the queries, since they preserve the object configuration, anyway). Moreover, the OCL pre-post conditions are translated into first-order

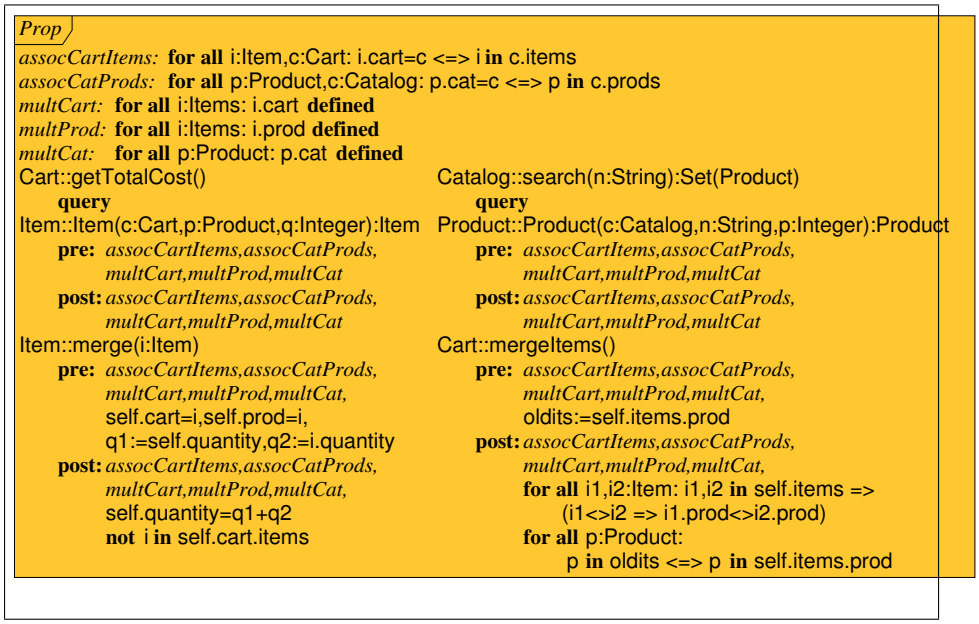


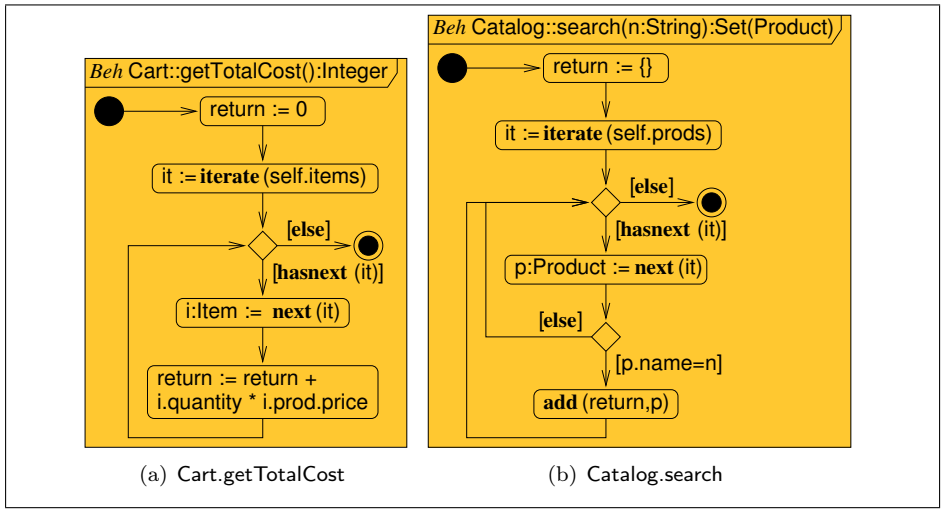
Figure 8: Descriptive low-level model of the example

logic, where the OCL @pre expressions are interpreted by using variables bound at pre time and reused at post time.

Now, the different behavioural techniques are translated into the low-level language, where constructive low-level models are visualised in the style of UML acitivity diagrams. But, while complex activity diagrams might employ a lot of features and "syntactic sugar" like e. g. complex object flows or OCL constraints as guards, the constructive low-level models may only use a very limited set of atomic actions, which we will see in the following examples.

In Fig. 9, the translations of the OCL body constraints are depicted. Since both operations iterate over a collection of objects, we need atoms to support this iteration. The actions **iterate**, **hasnext**, and **next** serve this purpose. When generating code from a low-level model, these should map relatively easy to iterator concepts on the target platform. The getTotalCost operation in Fig. 9(a) only uses assignments to the return variable and simple arithmetic calculations in addition to the iteration actions, while the search operation in Fig. 9(b) also uses atoms {} and **add** for manipulating a set of objects.

In Fig. 10, the translations of the transformation rules from Fig. 5 are given. Here, we see that the low-level language also supports parallelism. This can be employed when generating code for a target platform also supporting parallelising independent activities, like e. g. the .NET platform. The possible parallelism arises from the fact, that transformation rules do not prescribe a certain order, in which the changes to the object configuration shall be applied. The exact model transformation from the high-level transformation rules to the low-level language is out of the scope of this paper.



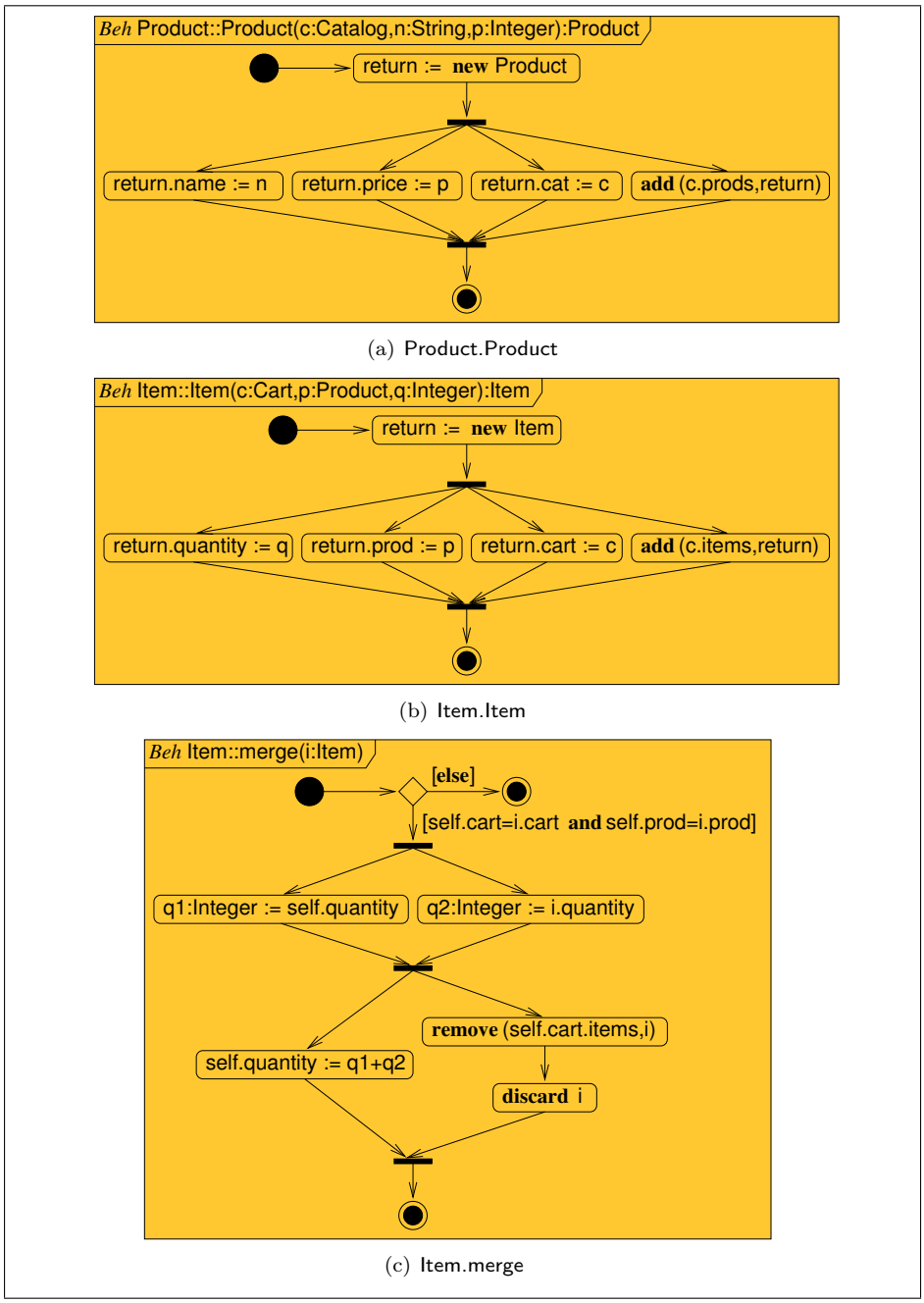
(a) Cart.getTotalCost (b) Catalog.search

Figure 9: Contructive low-level model for OCL constraints

The two constructor models in Fig. 10(a) and 10(b) make use of the atom **new**, which allocates a new uninitialised object. In Fig. 10(c), where the merge operation is modelled, we see how patterns in the left-hand side of a rule are translated into a decision constraining the applicability of the rule. Moreover, we see that attributes, which are accessed with an @pre notation in the right-hand side of a rule, are saved into variables prior to applying the changes. The merge operation also makes use of an additional atom remove for the manipulation of object sets and the **discard** atom for ending the life cycle of an object. The latter may be ignored, when generating code for a platform with garbage collection, but may be useful on other platforms like C++, where objects have to be destroyed explicitly.

Finally, in Fig. 11, the translation of the small flowchart from Fig. 6 is given. It uses two nested iterations over the same set and as a last kind of atomic action a call to another operation.

Now, we have seen the whole low-level model of our small example. As stated before, its purpose is the facilitation of code generation and verification. The suitability of the constructive model parts for code generation should be quite obvious, since the atoms used in the model are quite close to the basic instructions on common object-oriented platforms or the operations available on their collection implementations, respectively. For verification, a first approach could be the definition of a Hoare-like calculus, such that the per-operation requirements in the descriptive model could be verified along the structure of the corresponding operations.



(a) Product.Product

(b) Item.Item

(c) Item.merge

Figure 10: Contructive low-level model for transformation rules

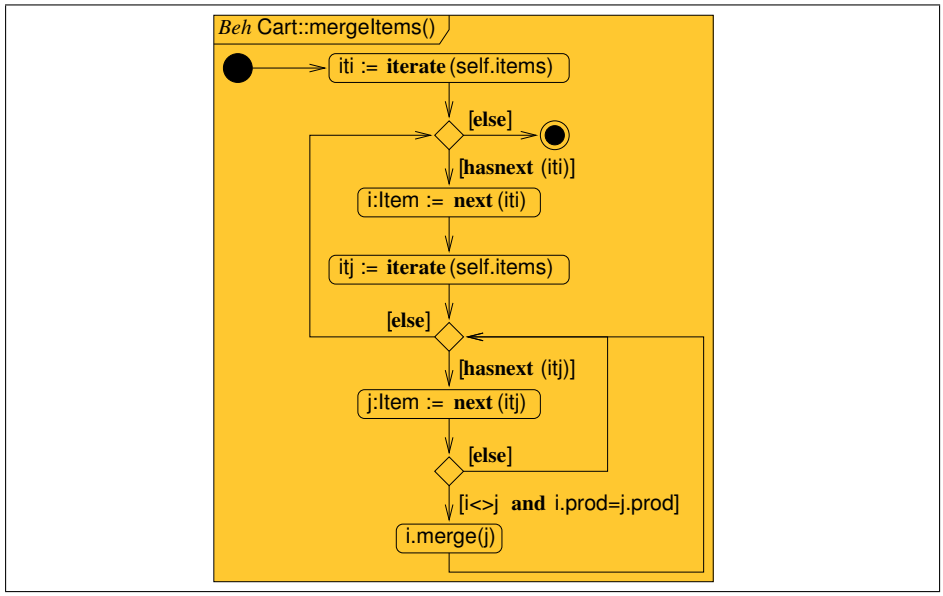


Figure 11: Constructive low-level model for flowchart

5 Summary, Related and Future Work

In this paper an abstract concept for the integration of complex multi-paradigm modelling techniques was proposed. It is build around the idea of translating complex, user-friendly models into a minimalistic, machine- and theory-friendly low-level language. This low-level language can be divided into structural, descriptive, and constructive elements, which is useful to ease code generation and verification.

As an instantiation of the concept, a complex modelling technique based on the UML and an object-oriented low-level language were sketched. The UML-based modelling technique uses behavioural techniques from different paradigms, namely functional OCL constraints, declarative transformation rules, and imperative flowcharts, while in the low-level language all these techniques are translated into the same style of low-level action flows.

A lot of formalisms have been proposed as rigorous foundations for complex modelling techniques. For example, a mapping from UML 1.3 activity diagrams to abstract state machines is proposed by Börger et al. (see [1]). Other approaches try to use process algebras as a semantic domain. While these proposals have the advantage of readily available verification and analysis tools, they need a lot of encoding to represent complex structures using the means of the algebraic formalisms, which reduces the intuition behind the translations. Moreover, the approach in this paper targets generation of code for the behavioural models, where algebraic formalisms would be a detour, given that UML activities already have a rather imperative structure.

In [9], Störrle and Hausmann evaluate the possibility to use Petri nets as a semantic domain for UML activity diagrams, which is also suggested by the UML specification itself. They come to the conlusion, that, in order to integrate all possibilities of activity diagrams, different variants of Petri nets would have to be integrated in a new formalism, which would then have neither tools nor theory available. This observation may also serve as a reason for deriving the new low-level language proposed in this paper, which is specifically designed to capture the features of complex object-oriented systems.

A first point of future work is the establishment of meta-models for both CUML and the low-level language and the development of a tool-supported model transformation between these models, which implements the translation sketched in this paper. Furthermore, the implementation of a proof-of-concept code generator for the low-level language is planned.

In order to retain compatibility with the widely used UML standard and other UML tools, we will try to formalise CUML as a UML profile, so that class diagrams, OCL expressions and activity diagrams are restricted to the subclass we consider, and transformation rules and flowcharts are realised as concrete syntaxes for special kinds of UML activities.

On the theoretical side, a formal semantics for the low-level language will be developed, which will also allow to reason about compositionality of low-level and complex models. For this purpose, concepts of visibility and imports of model elements will be introduced into the modelling techniques. A formal semantics will also allow the development of formal refinements and refactorings of models.

Moreover, formal analysis methods and tools should be developed for the low-level language, where exisiting work on verification techniques for graph transformation systems could serve as a basis, since the formal semantics will execute the low-level models by rules, which are very similar to graph rewriting rules.

Finally, the extension of the complex modelling technique with domain-specific extensions is an interesting line of future research. These extensions should be possible rather easily, because of the modular structure of the approach. The low-level language can be left unchanged and the new domain-specific language only has to be translated into this fixed low-level language, where new domain-specific languages can either be translated into corresponding UML diagrams providing for an indirect integration, formulated as an additional UML profile with its own translation into the low-level language, or equipped with a meta-model independent of the UML meta-model.

References

[1] Börger, E., A. Cavarra and E. Riccobene, *An ASM semantics for uml activity diagrams*, in: *Algebraic Methodology and Software Technology, AMAST 2000*, LNCS **1816**, Springer, 2000 pp. 293–308.

[2] Braatz, B., *A rule-based, integrated modelling approach for object-oriented systems*, in: *Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*, ENTCS (2006), to appear.

[3] Braatz, B. and A. R. Kniep, *Integration of object-oriented modelling techniques* (2006), draft version available from `http://tfs.cs.tu-berlin.de/~bbraatz/papers/BK06-TR.pdf`.

[4] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation," Monographs in Theoretical Computer Science, Springer, 2006.

[5] Große-Rhode, M., "Semantic Integration of Heterogeneous Software Specifications," Monographs in Theoretical Computer Science, Springer, 2004.

[6] Nassi, I. and B. Shneiderman, *Flowchart techniques for structured programming*, ACM SIGPLAN Notices **8** (1973), pp. 12–26, `http://www.geocities.com/SiliconValley/Way/4748/nsd.html`.

[7] Object Management Group, "UML Superstructure Specification, v2.0," (2005), `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.

[8] Object Management Group, "Object Constraint Language, v2.0," (2006), `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

[9] Störrle, H. and J. H. Hausmann, *Towards a formal semantics for UML 2.0 activities*, in: *Software Engineering, SE 2005*, 2005, available from `http://www.pst.informatik.uni-muenchen.de/~stoerrle/`.

Author Index