

Mehmet Aksit
Ekkart Kindler
Ashley McNeile
Ella Roubtsova (Eds.)

Behaviour Modelling in Model Driven Architecture

First European Workshop
on Behaviour Modelling in Model Driven Architecture
(BM-MDA)
Enschede, The Netherlands, June 23, 2009
Proceedings



Enschede, the Netherlands, 2009
CTIT Workshop Proceedings Series WP09-04
ISSN 0929-0672

Organisation

Organizing Committee

Mehmet Aksit.	TU Twente, the Netherlands
Ekkart Kindler	Technical University of Denmark
Ashley McNeile	Metamaxim Ltd, UK
Ella Roubtsova	Open University of the Netherlands

Programme Committee

Mehmet Aksit.	TU Twente, the Netherlands
Michael Jackson	Open University, UK
Ekkart Kindler	Technical University of Denmark
Reiko Heckel	University of Leicester, UK
Dominik Stein	University of Duisburg-Essen, Germany
Luis Gomes	Universidade Nova de Lisboa, Portugal
Ashley McNeile	Metamaxim Ltd, UK
Louis Birta	University of Ottawa, Canada
Stewart Robinson	University of Warwick, UK
João M. Fernandes	Universidade do Minho, Portugal
Ella Roubtsova,	Open University of the Netherlands
Stefan Hanenberg	University of Duisburg-Essen,Germany

Table of Contents

Keynote:

Automatic generation of behavioral code – too ambitious or even unwanted?
 G. Engels, University of Paderborn, Germany

Preface.....	6
M.Aksit,University of Twente,Enschede, the Netherlands	
E. Kindler,Technical University of Denmark, Copenhagen	
A. McNeile, Metamaxim Ltd,London, UK	
E. Roubtsova, Open University of the Netherlands	
1. Weaving executability into UML class models at PIM level.....	
E. Riccobene University of Milan, Italy	
P. Scandurra University of Bergamo, Italy	
2. Behaviour Modelling Notation for Information System Design.....	
A. Kalnins, E. Celms, E. Kalnina and A. Sostaks	
Institute of Mathematics and Computer Science	
University of Latvia	
3. Composition Semantics for Executable and	
Evolvable Behavioral Modeling in MDA	
A. McNeile, Metamaxim Ltd, UK	
E. Roubtsova, Open University of the Netherlands	
4. Towards a Model Execution Framework for Eclipse.....	
M. Soden and H. Eichler.	
Humbold University, Germany	
5. Towards Model Structuring Based on Flow Diagram	
A. Rensink and M. Zimakova,	
University of Twente, the Netherlands	
6. Recursive Modeling for Completed Code Generation.....	
S. Sulistyo and A.Prinz	
University of Agder, Norway	
7. Embedding Process Models in Object-oriented Program Code.....	
M. Balz and M Goedicke	
University of Duisburg-Essen, Germany	

Keynote: Automatic generation of behavioral code - too ambitious or even unwanted?

Gregor Engels

University of Paderborn, Warburger Str. 100 33098 Paderborn Germany
engels@upb.de

Abstract. Pushing the button and yielding automatically generated code from behavioral models is one of the dreams of the Model-Driven Architecture approach. Current practice shows that this works quite well in certain domain-specific contexts or in case of low-level (textual) behavior specifications. But, in most cases, behavioral models are only used in early phases of a software development project for documenting user requirements. They are not used for an automatic code generation. This also leads to the problem that behavioral models are separated from the code, which finally even leads to dead models. The talk discusses the role of behavioral models within software development processes. It addresses and compares several approaches for behavior modeling (e.g. (visual) pre-/post-conditions, protocol specifications, graph transformations) and discusses importance and limitations of an automatic code generation. A discussion on alternative usages of behavioral models like model-driven testing, model-driven monitoring and models@runtime concludes the talk.

Preface

Mehmet Aksit¹, Ekkart Kindler², Ashley McNeile³ and Ella Roubtsova⁴

¹ University of Twente, Enschede, the Netherlands, m.aksit@ewi.utwente.nl

² Technical University of Denmark, Copenhagen, Denmark, eki@imm.dtu.dk

³ Metamaxim Ltd, London, UK, ashley.mcneile@metamaxim.com

⁴ Open University of the Netherlands, ella.roubtsova@ou.nl

BM-MDA is the first International Workshop on Behaviour Modelling in Model Driven Architecture organized in Enschede (the Netherlands) on the 23d of June 2009 in conjunction with the European Conference on Model Driven Architecture.

The Model-driven Architecture (MDA) features the use of different kinds of models during the software development process and automatic transformations between them. One of the main ideas is the separation between models that are platform independent (PIMs) and models that are platform specific (PSM). From these models, some parts of the final code can be automatically generated. Ultimately, the goal is to generate the complete software fully automatically from these models. The purpose of the BM MDA workshop is to better understand what is needed to adequately model behaviour in MDA, and what is still lacking for universally modelling the behaviour of software in such a way that the code can automatically be generated from them.

The papers accepted for the workshop show that fully automatic generation of the code from models is still a dream and, if it works at all, restricted to specific application areas. The contributions describe various approaches to providing better support for the aims of MDA, including

- Various changes to the semantics and notations of UML behaviour modelling techniques;
- The introduction of new modelling semantics;
- Improved approaches to providing execution semantics for behavioural models.

Several papers in the program of the workshop propose to extend or replace the semantics of the behavioural diagrams in the UML in order to achieve executability of models.

”Weaving executability into UML class models at PIM level” by E. Riccobene and P. Scandurra describes an approach to extending UML with ASM (Abstract State Machine) behaviour and execution semantics. The weaving is achieved using the identification of join-points between the UML and ASM metamodels, rather in the manner of aspect weaving, resulting in a combined language the authors call UML+. The authors show how the semantics of the UML+ meta-model may be defined in terms of the semantics of the component metamodels and illustrate the approach by adding ASM behaviour semantics to the UML Class Diagram. This is an attractive idea, as it allows existing behavioural formalisms (and their related tools, in particular for execution/simulation and code

generation) to be used within the context of UML modelling. However, achieving a weaving with the Class Model is the easy part, as there is no "semantic overlap", and the real test of the approach will be to achieve sensible results with undue complexity when considering more general Class behaviour, including inter-object communication. Another challenge is to see how the reasoning possibilities afforded by such behavioural models as ASM can be used in the context of UML models.

"Behaviour Modelling Notation for Information System Design" by A. Kalnins, E. Celms, E. Kalnina, and A. Sostaks describes work on extending two UML behavioural modelling notations, Sequence Diagrams and Activity Diagrams, to improve expressive power. The suggested changes for Sequence Diagrams are minor and relate primarily to loop constructs. The suggested changes for Sequence Diagrams are more radical, and aim to give Sequence Diagrams the power to represent behaviour involving multiple classes, as Sequence Diagrams do, using a "swimlane" concept. The result is a notation similar in concept to Sequence Diagram, but using a different approach to designate the sequencing of execution. The work on the approach is in the progress as it does not show how the interactions between all classes can be presented, how many Sequence Diagrams should be used to specify a system, and what is the semantics of the composition of multiple Diagrams (particularly where the same Class appears in more than one diagram).

"Composition Semantics for Executable and Evolvable Behavioral Modeling in MDA" by A. McNeile and E. Roubtsova analyses the UML's two forms of state machine (Behaviour State Machines and Protocol State Machines) and concludes that neither of them is particularly suitable for behaviour composition and evolution of models. Instead, a new protocol modelling semantics using CSP parallel and CCS composition operators is suggested. This is a radical departure from the current UML semantics, as it promotes process algebraic behavioural composition to first class status in the modelling language. This semantics gives advantages for local reasoning, based on the established reasoning techniques of process algebra, and allows a compositional style of behaviour model construction and re-use.

"Towards a Model Execution Framework for Eclipse" by M. Soden and H. Eichler describes work to extend the family of Eclipse modelling tools with a Model Execution Framework (MXF). The aim of this work is to provide a toolset that supports the building of execution capabilities in the Eclipse modelling environment, and provides a language (MAction Language) that is claimed to provide a set of basic actions from which execution DSLs can be built. The use of MXF is illustrated by showing how it can be used to build a StateMachine execution capability. The XMF facility aims to be a general kit for building behavioural languages and not restrictive on the semantics of the behavioural language being equipped for execution. This is potentially a powerful tool. The authors do not discuss whether their language has theoretical limits to its capabilities, and this may be an interesting area for further investigation.

Two papers concern approaches to model transformation and refinement for integration, re-integration and execution of models.

”Towards Model Structuring Based on Flow Diagram Decomposition” by A. Rensink and M. Zimakova presents an experimental implementation of decomposition algorithms for transforming, by means of graph transformation, unstructured flow graphs into structured ones. Such algorithms have been studied in theory (e.g., in order to prove that both classes have the same formal expressiveness), but are not yet well-studied in practice in terms of their complexity and implementation. The motivation is model understanding as well as implementation of general behaviour models in structured target languages. A number of methods have been implemented with graph transformations, employing the graph-transformation tool Groove for rule execution. The complexity measure of a flow graph was developed to reflect an intuitive notion of flow graph reliability and readability and evaluate different decomposition algorithms and results of non-deterministic algorithms for the purpose of the complexity minimization.

”Recursive Modeling for Completed Code Generation” by S. Sulistyo and A. Prinz gives an example of recursive refinement of structural and behavioural models in order to achieve executability. The authors present an interesting general discussion of modelling, including structural and behavioural models, high-level models and executable elements. They propose recursive refinement of activity elements in activity diagrams until the mapping into existing components can be obtained. Refinement of activities is well known in UML. However, in this approach activities are transformed into classes. This concurrent refinement of activities embedded into structural models is not standard. Ideas of this approach appear in some work but have not been fully developed yet. Some elements of this approach need further investigation. For example, the notion of consistency of behavioural and structural diagrams needs to be defined. Manual refinement is error prone and needs procedures for correctness checks. Moreover, presentation of interactions between concurrent activities has to be addressed. This can be seen as future work in development of this promising approach.

”Embedding Process Models in Object-oriented Program Code” by M. Balz and M. Goedicke presents an interesting idea on how process models can be embedded into object-oriented program code by means of a design pattern. Though this approach is still more focussed on programming, it builds a bridge between the classical way of software development by programming and the upcoming model-based approaches, by defining a clear interface between process models and programs. This way, it takes a first steps to solving one of the main problems with behaviour modelling today: the integration of behaviour models with other models and with pre-existing program code. It is yet to be seen how far this specific solution will carry; but the ideas behind that pattern could probably also be used for integrating other behaviour models. For now, the pattern eases combining process models from the Java Workflow Tooling project (JWT) into program code.

Analysis of the papers shows that the problem of executable modelling at the Platform Independent Level still awaits a standard and elegant solution. Current

UML behavioural modelling semantics does not properly meet the challenge that MDA presents and various extensions or replacements have been proposed. A simple compositional semantics that supports reasoning about models and allows transformation into the code is a critical factor for the success of MDA, and we believe that the ideas contributed to this workshop will help show the way towards achieving the goals of MDA.

The organizers of BM-MDA thank all the authors for their contribution and the members of the Program Committee for their excellent reviews of the submitted papers.

Weaving executability into UML class models at PIM level

Elvinia Riccobene² Patrizia Scandurra¹

¹ DTI - Università degli Studi di Milano, Italy

elvinia.riccobene@unimi.it

² DIIMM - Università degli Studi di Bergamo, Italy

patrizia.scandurra@unibg.it

Abstract. Modeling languages that aim to capture PIM level behavior are still a challenge. We propose a high level behavioral formalism based on the Abstract State Machines (ASMs) for the specification and validation of software systems at PIM level. An ASM-based extension of the UML and its Action Semantics is here presented for the construction of executable class models at PIM level and also a model weaving process which makes the execution of such models possible. Our approach is illustrated using an Invoice Order System taken from the literature.

1 Introduction

Model-driven Engineering (MDE)[3] promotes *models* as first class artifacts of the software development process and automatic *model transformations* to drive the overall design flow from requirements elicitation till final implementations toward specific platforms. Model Driven Architecture (MDA) [25], which supports various standards including the UML (Unified Modeling Language) [37], from the OMG (Object Management Group) is the best known MDE initiative.

In the MDA context, notations usually based on the UML are used as system modeling languages for producing *platform-independent models* (PIMs) and *platform-specific models* (PSMs). Automatic model transformations allow transforming PIMs into PSMs.

Although MDE frameworks (OMG/MOF, Eclipse/Ecore, GME/MetaGME, AMMA/ KM3, XMF-Mosaic/Xcore, etc.) are currently able to cope with most syntactic and transformation definition issues, *model executability* is still remarked as a challenge [27], especially at PIM level. One of the main obstacles is the lack of adequate models for the behavior of the software and of mechanisms to integrate behavioral models with structural models and with other behavioral models. Although there are many different approaches for modeling behavior (see related work in Sect. 2), none of them enjoys the same universality as the UML class diagrams do for the structural parts of the software. Further evidence of confusion about PIM level behavioral modeling is the lack of agreement on what basic behavioral abstractions are required, and how these behavioral abstractions should be used. However, PIM executability is considered a remarkable

feature for the system development process, since it allows verifying high-level models against the requirements goals (possibly using automated analysis tools), and it can be exploited to provide conformance for implementations at PSM and code level by generating test-cases.

A current crucial issue in the MDA context is, therefore, that of providing effective specification and validation frameworks able to express the meaning or semantics of each modeling element and interaction occurring among objects, rather than dealing with behavioral issues depending on the target implementation platform. We believe this goal can be achieved by integrating MDE/MDA structural modeling notations with a behavioral formalism having the following features: (i) it should be abstract and formal to rigorously define model behavior at different levels of abstraction, but without formal overkill; (ii) it should be able to capture heterogenous models of computation (MoC) in order to smoothly integrate different behavioral models; (iii) it should be executable to support model validation; (iv) it should be endowed with a model refinement mechanism leading to correct-by-construction system artifacts; (v) it should be supported by a set of tools for model simulation, testing, and verification; (vi) it should be endowed with a metamodel-based definition in order to exploit MDE techniques of automatic model transformations.

In this paper, we address the issue of providing executability to PIMs by using the ASM (Abstract State Machine) [6] formal notation that owns all the characteristics of preciseness, abstraction, refinement, executability, metamodel-based definition, that we identified above as the desirable properties for this goal. We propose an ASM-based extension of the UML and its Action Semantics to define a high level behavioral formalism for the construction of executable PIMs. This is achieved by *weaving* behavioral aspects expressed in terms of ASM elements into the UML metamodel. The ASM formal notation becomes, therefore, an abstract action language for UML at PIM level, and, by automatic models mapping, we are able to associate an ASM executable model to a UML model. In particular, we apply our technique to the UML class diagrams since they are considered to be the standard for modeling the structural parts of software. The approach is anyway applicable to any other part of the UML metamodel and to any modeling language whose abstract syntax is given in terms of a metamodel.

This paper is organized as follows. Sect. 2 provides a description of related work along the lines of our motivation. Some background concerning the ASMs is given in Sect. 3. Sect. 4 presents the proposed weaving approach between the UML and the ASM metamodels in order to provide a high level formalism to specify behavior at PIM level. In Sect. 5 we define how to automatically map UML class models into executable ASM models and the action semantics provided to the UML class diagrams by the ASM elements. Implementation details about the automatic model transformation are given in Sect. 6. Sect. 7 presents the application of our UML/ASM-based modeling notation to the Invoice Order System case study. Finally, Sect. 8 concludes the paper and outlines some future directions of our work.

2 Related Work and Motivation

There are different approaches for modeling and executing behavior in the UML at PIM level. They may mainly fall into the following categories.

(I) *Not include behavior in the PIM at all*, but instead add it as code to structural code skeletons later in the MDA process. This, however, prevent us from making significant early validation of the system.

(II) *Provide preliminary executability at meta-language level*. Some recent works (like Kermeta [29], xOCL (eXecutable OCL) [38], or also the approach in [35], to name a few), have addressed the problem of providing executability into current metamodeling frameworks like Eclipse/Ecore [13], GME/MetaGME [19], XMF-Mosaic/Xcore [38], etc. This approach is merely aimed at specifying the semantics of a modeling language (another key current issue for model-based engineering) and thereby at providing techniques for semantics specification natively with metamodels.

(III) *Use the OCL* [31] (and its various extensions, see [9] for example) to add behavioral information (such as pre- and post-conditions) to other, more structural, UML modeling elements; however, being side-effect free, the OCL does not allow the change of a model state, though it allows describing it.

(IV) *Joint use of an action language*, based on the UML action semantics (AS) [37, 14], and of UML behavioral diagrams such as state machines, activity diagrams, sequence diagrams, etc., possibly strengthening their semantics. Behavioral diagrams can be used to capture complete behavioral information as part of the PIM. The UML AS use a minimal set of executable primitives (create/delete object, slot update, conditional operators, loops, local variables declarations, call expressions, etc.) to define the behavior of metamodels by attaching behavior to classes operations (the specification of the body counterpart is usually described in text using a surface action language). Probably the most well-known example is the approach known as “Executable UML (xUML)” [32]). Recently, a beta version [14] has been released of an executable subset of the standard UML (the *Foundational UML Subset*) to be used to define the semantics of modeling languages such as the standard UML or its subsets and extensions, and therefore providing a foundation for the definition of a UML virtual machine capable of executing UML models.

(V) *Transform UML diagrams into formal models*; e.g. transform class and sequence diagrams into graphs [11] by using graph-transformation rules in order to create a set of graphs that represent the state-space of the behavior, and then apply model checking techniques on this state-space to verify certain properties of the UML models. Similar approaches based on this *translational* technique are UML-B [36] using the Event-B formal method, those adopting Object-Z like [26, 28], etc.

The approach proposed in this paper is slightly different from the above ones. The objective is to provide a behavioral formalism at PIM level that does not depend on a particular UML behavioral diagram, since it should be general enough for other metamodel-based languages not necessarily related to the UML. Moreover, in terms of expressiveness, non-determinism and executability (as ASMs

support) are two important features to be taken into account for the specification and validation of the behavior of distributed applications and application components.

The ASMs formalism itself can be also intended as an action language but with a concise, abstract and powerful set of action schemes. That allows to overcome some limits of conventional action languages based on the UML AS. These last, – though they aim to be pragmatic, extensible and modifiable – may suffer from the same shortcomings and complexity of traditional programming languages being too much platform-specific.

Moreover, not all action semantics proposals are powerful enough to reflect a particular model of computation (MoC) underlying the nature of the application being modeled. This, instead, is not true for the ASMs.

Through several case studies, ASMs have shown to be a formal method suitable for system modeling and, in particular, for describing the semantics of modeling/programming languages. Among successful applications of the ASMs in the field of language semantics, we can cite the UML and SDL-2000, programming languages such as Java, C/C++, and hardware description languages (HDLs) such as SystemC, SpecC, and VHDL – complete references can be found in [6]. Concerning the ASM application to provide an executable and rigorous UML semantics, we can mention the works in [30, 5, 7, 23, 10]. More or less, all these approaches define an ASM model able to capture the semantics of a particular kind of UML graphical sub-language (statecharts, activity diagrams, etc.). Other attempts in this direction but generalized to any metamodel-based language – and therefore belonging to category (II) – are the works in [8, 12], to name a few. However, the use of the ASMs we suggest here is different. Here we focus on the use of the ASMs as “modeling language” (at the same level of the UML) rather than as “meta-language” (or semantics specification language). The goal is to provide a general virtual machine at PIM level by “weaving” executable behavior directly into structural models.

3 Abstract State Machines

Abstract State Machines (ASMs) are an extension of FSMs [4], where unstructured control states are replaced by states comprising arbitrary complex data.

Although the ASM method comes with a rigorous mathematical foundation [6], ASMs provides accurate yet practical industrially viable behavioral semantics for pseudocode on arbitrary data structures. This specification method is tunable to any desired level of abstraction, and provides rigor without formal overkill.

The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them, while the *transition relation* is specified by “rules” describing how functions change from one state to the next.

Basically, a transition rule has the form of *guarded update* “**if Condition then Updates**” where *Updates* are a set of function updates of the form $f(t_1, \dots, t_n) := t$ which are simultaneously executed³ when *Condition* is true.

These is a limited but powerful set of *rule constructors* that allow to express simultaneous parallel actions (**par**) of a single agent, either in an atomic way, *Basic ASMs*, or in a structured and recursive way, *Structured or Turbo ASMs*, by sequential actions (**seq**), iterations (**iterate**, **while**, **recwhile**), and submachine invocations returning values. Appropriate rule constructors also allow non-determinism (existential quantification **choose**) and unrestricted synchronous parallelism (universal quantification **forall**). Furthermore, it supports a generalization where multiple agents interact in parallel in a synchronous/asynchronous way, *Synchronous/Asynchronous Multi-agent ASMs*.

Based on [6], an ASM can be defined as the tuple:

$(\text{header}, \text{body}, \text{main rule}, \text{initialization})$

The *header* contains the *name* of the ASM and its *signature*⁴, namely all domain, function and predicate declarations. Function are classified as *derived* functions, i.e. those coming with a specification or computation mechanism given in terms of other functions, and *basic* functions which can be *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are further classified into: *monitored* (only read, as events provided by the environment), *controlled* (read and write), *shared* and *output* (only write) functions.

The *body* of an ASM consists of (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations (definitions) of transition rules. The body of ASM may also contains definitions of *axioms* for invariants one wants to assume for domains and functions of the ASM.

The (unique) *main rule* is a transition rule and represents the starting point of the machine program (i.e. it calls all the other ASM transition rules defined in the body). The main rule is *closed* (i.e. it does not have parameters) and since there are no free global variables in the rule declarations of an ASM, the notion of a move does not depend on a variable assignment, but only on the state of the machine.

The *initialization* of an ASM is a characterization of the initial states. An initial state defines an initial value for domains and functions declared in the signature of the ASM. *Executing* an ASM means executing its main rule starting from a specified initial state. A *computation* of M is a finite or infinite sequence $S_0, S_1, \dots, S_n, \dots$ of states of M , where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing simultaneously all of the transition rules which are enabled in S_n .

³ f is an arbitrary n -ary function and t_1, \dots, t_n, t are first-order terms. To fire this rule to a state S_i , $i \geq 0$, evaluate all terms t_1, \dots, t_n, t at S_i and update the function f to t on parameters t_1, \dots, t_n . This produces another state S_{i+1} which differs from S_i only in the new interpretation of the function f .

⁴ *Import* and *export* clauses can be also specified for modularization.

3.1 The ASM Metamodel and ASMETA

In addition to its mathematical-based foundation, a metamodel-based definition for ASMs is also available. The ASM metamodel, called *AsmM* (*Abstract State Machines Metamodel*) [33, 15, 17, 2], provides an abstract syntax for an ASM language in terms of MOF concepts, and has been defined with the goals of developing a *unified* abstract notation for the ASMs, independent from any specific implementation syntax and allowing a more direct encoding of the ASM mathematical concepts and constructs.

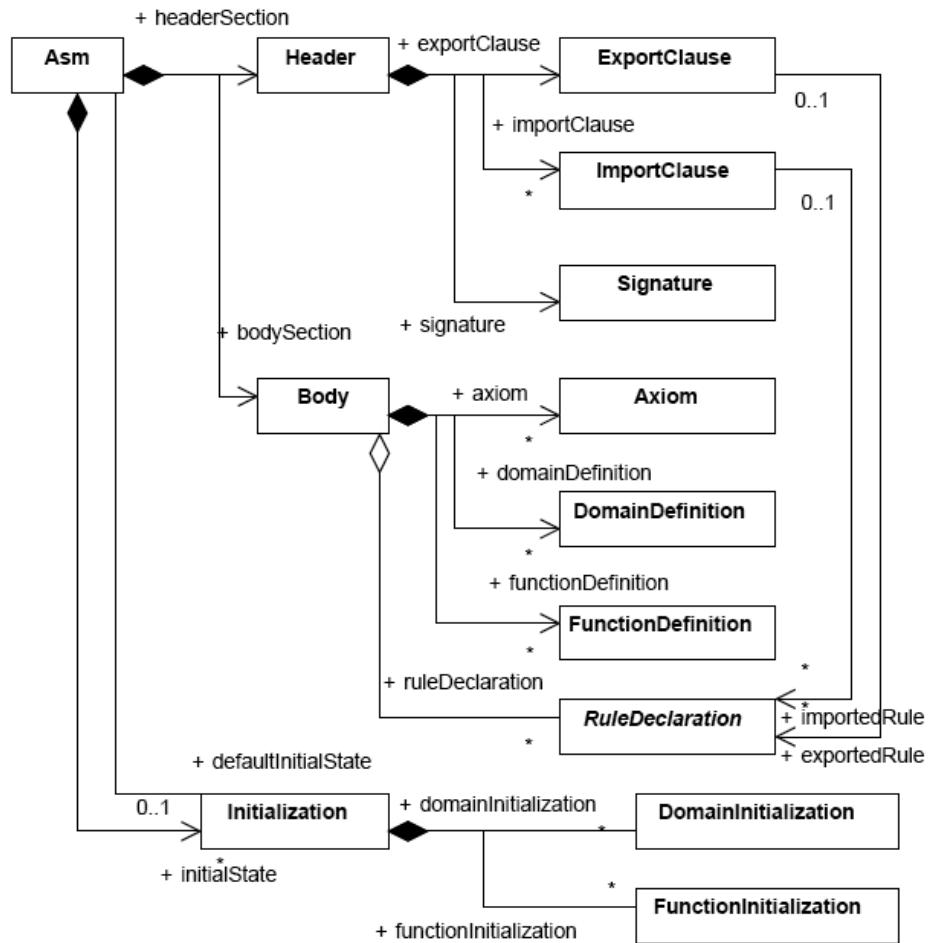


Fig. 1. Backbone

Fig. 1 shows a very small fragment of the *AsmM* metamodel representing the structure of an ASM model.

AsmM is publicly available (see [2]) in the meta-language EMF/Ecore [13].

The AsmM semantics was given by choosing a semantic domain S_{AsmM} and defining a *semantic mapping* $M_S : AsmM \rightarrow S_{AsmM}$ to relate syntactic concepts to those of the semantic domain. S_{AsmM} is the first-order logic extended with the logic for function updates and for transition rule constructors formally defined in [6].

A general framework, called *ASMETA tool set* [16, 2], has been developed based on the AsmM and exploiting the advantages of the metamodelling techniques. It essentially includes: a textual notation, *AsmetaL*, to write ASM models (conforming to the AsmM) in a textual and human-comprehensible form; a text-to-model compiler, *AsmetaLc*, to parse ASM models written in AsmetaL and check for their consistency with respect to the OCL constraints of the metamodel; a simulator, *AsmetaS*, to execute ASM models (stored in a model repository as instances of AsmM); the *Avalla* language for scenario-based validation of ASM models, with its supporting tool, the *AsmetaV* validator; the *ATGT* tool that is an ASM-based test case generator based upon the SPIN model checker; a graphical front-end, called *ASME* (ASM Eclipse Environment), which acts as IDE and it is an Eclipse plug-in.

4 Weaving executable behavior into UML Class Model

The aim of this technique is to weave behavioral aspects into the whole UML metamodel or parts of it, depending on those elements one is interested to express behavior about.

Applying this technique demands the definition of a *weaving function* specifying how the UML metamodel and the AsmM are weaved together into a new metamodel which adds to the UML the capability of specifying behavior by ASM transition rules. More precisely, it requires identifying precise *join points*⁵ between data and behavior [29], to express how behavior can be attached to structural constructs.

Once a weaving function has been established between the UML and the AsmM, the resulting metamodel, in the sequel referred as *UML⁺*, enriches the UML with behavior specification capability in terms of ASM transition rules. Therefore, *UML⁺* can be considered an abstract structural and executable language at PIM level.

As example of weaving executable behaviors into structural models by using ASMs, we here consider the portion of the UML metamodel concerning with class diagrams. However, the weaving process described here is directly applicable to any object-oriented metamodel and meta-metamodel like the OMG MOF, EMF/ECore, AMMA/KM3, etc.

⁵ Inspired from the Aspect-oriented Programming (AOP) paradigm, join points are intended here and in [29] as places of the meta-metamodel where further (executability) aspects can be injected.

4.1 Join Points Identification

In case of the UML metamodel, as for any other MOF metamodel, it might be convenient to use transition rules within meta-classes as class operations to hold their behavioral specification. Therefore, a join point must be specified between the class **Operation**⁶ of the UML (see Fig. 7.11 in [37]) and the class **RuleDeclaration** of the AsmM.

Fig. 2 shows how simply the composition may be carried out. The MOF **Operation** class resembles the AsmM **RuleDeclaration** class. The name **Operation** has been kept instead of **RuleDeclaration** to ensure UML conformance; similarly, the name **Parameter** has been kept instead of **VariableTerm**. Finally, the new property **isMain** has been added in order to designate, when set to true, a *closed* (i.e. without formal parameters) operation as (unique) main rule of an *active* class (the *main* class) to start model execution.

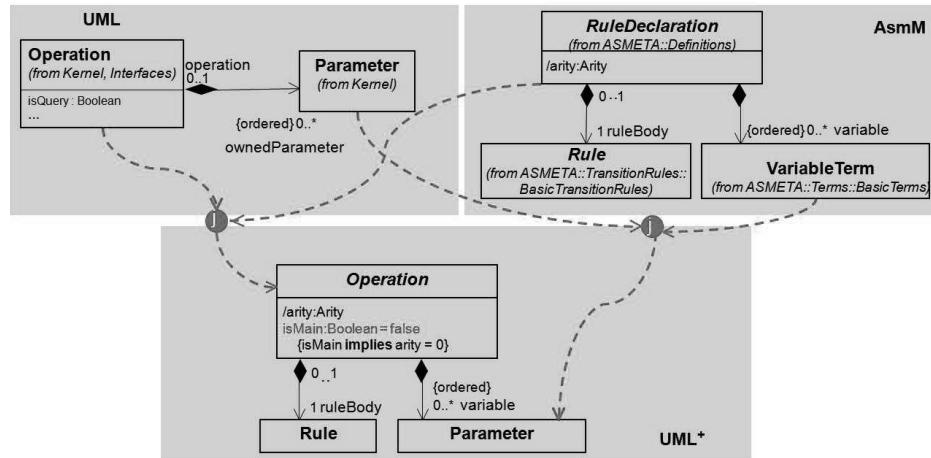


Fig. 2. Using operation bodies as join points between data and behaviour

A further join point is necessary to adorn UML class's properties (either attributes or association member ends – see Fig. 7.12 in [37]) to reflect the ASM function classification. Fig. 3 shows how this may be carried out. The UML **Property** class resembles the AsmM **Function** class. Box UML⁺ presents the result of the composition process. The UML class **Property** has been merged with the class **Function**. A further adornment **kind:PropertyKind** have been added to capture the complete ASM function classification. **PropertyKind** is an enumeration of the following literal values: static, monitored, controlled, out, and shared. Two OCL constraints have been also added stating, respectively, that a *read-only* (attribute `isReadOnly` is set to true) property can be of kind

⁶ An operation is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior.

static or monitored, and that if a property is *derived* (attribute `isDerived` is set to true) then the attribute `kind` is empty.

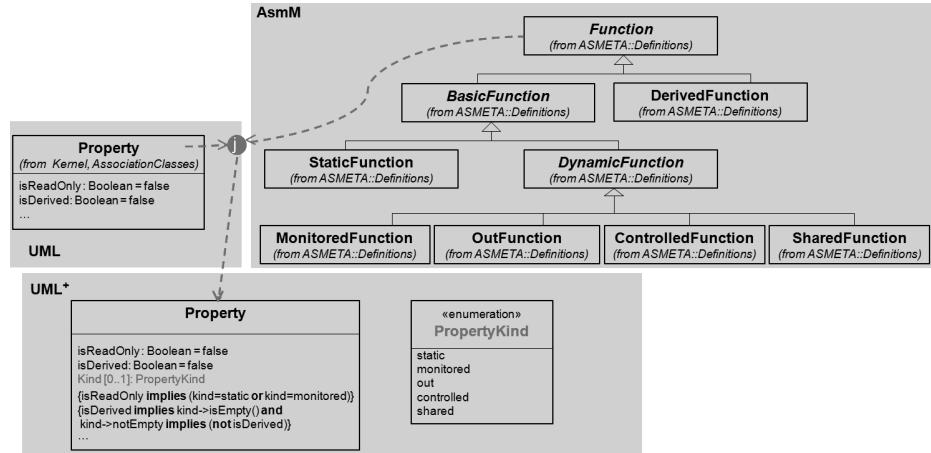


Fig. 3. Using properties as join point for ASM adornments

Moreover, in order to merge the two statically-typed systems of the UML and the AsmM, a UML Type (a `Class` or a `DataType`) is merged with an ASM Domain. Finally, values specification in UML class models (e.g. for specifying default values of attributes) are provided in terms of *opaque expressions* (instances of the `OpaqueExpression` class in the UML metamodel)⁷ that are merged with ASM terms (the `Term` class of the AsmM metamodel).

5 Semantic Model

At this point of the weaving process, we are able to design by the `UML+` terminal models [24] whose syntactic elements conform to UML and whose operation semantics is expressed in terms of ASM rules. (Sect. 7 reports an example of application). So doing, the ASM formal notation can be considered as an abstract action language for UML at PIM level. However, following our approach, we are able to provide more, namely to define in a precise and clear way the executable semantics of a terminal model conforming to `UML+`, by associating it with its ASM semantic (executable) model. This is clarified by the following argumentation that refers to a generic metamodel (more details can be found in [18]), but which is then tailored for the `UML+` in Sect. 5.1.

A language metamodel A has a well-defined semantics if a semantic domain S is identified and a semantic mapping $M_S : A \rightarrow S$ is provided [21] to give

⁷ In UML, an opaque expression is an uninterpreted textual statement that denotes a (possibly empty) set of values when evaluated in a context

meaning to syntactic concepts of A in terms of the semantic domain elements. By exploiting the ASM formal method endowed with a metamodel representation of their concepts and with a precise mathematical semantics, we can express the semantics of a terminal model [24] conforming to A in terms of an ASM model. Let us assume the semantic domain S_{AsmM} of the ASM metamodel (see Sect. 3.1) as the semantic domain S . The semantic mapping $M_S : A \rightarrow S_{AsmM}$, which associates a well-formed terminal model m conforming to A with its semantic model $M_S(m)$, can be defined as

$$M_S = M_{S_{AsmM}} \circ M$$

where $M_{S_{AsmM}} : AsmM \rightarrow S_{AsmM}$ is the semantic mapping of the ASM metamodel and associates a theory conforming to the S_{AsmM} logic with a model conforming to $AsmM$, and the function $M : A \rightarrow AsmM$ associates an ASM to a terminal model m conforming to A . Therefore, the problem of giving the metamodel semantics is reduced to define the function M between metamodels. The complexity of this approach depends on the complexity of building the function M . In the following section, we show how to build the function M for the UML⁺ metamodel.

5.1 Semantic Model of UML⁺

The building function $M : UML^+ \rightarrow AsmM$ is defined as

$$M(m) = \iota(W(m), m)$$

for all terminal model m conforming to UML^+ , where:

- $W : UML^+ \rightarrow AsmM$ maps a weaved terminal m conforming to UML^+ into a model conforming to the $AsmM$ and provides the abstract data structure (signature, domain and function definitions, axioms) and the transition system of the final machine $M(m)$;
- $\iota : AsmM \times UML^+ \rightarrow AsmM$ computes the initial state of the final machine $M(m)$ by extracting initial values for data structures of the machine from the source modeling elements in m .

The function W is defined as a mapping in Table 1 and provides semantics both to the basic modeling elements characterizing UML class models (although we apply some restrictions as remarked below) and to the enriched UML⁺ modeling elements as result of the weaving process. The semantics is given by associating each modeling concept into a corresponding $AsmM$ modeling element. Below, we comment those parts of W concerning UML elements involved into the join points definition, while we leave to the reader intuition the understanding of the remaining parts.

The **Operation** element of the weaved language UML⁺ is associated to the corresponding **RuleDeclaration** element of the **AsmMas** involved in the join point definition. We assume, similarly to the use of **this** in the Java programming language, that in the definition of the rule body of an operation op , a special variable named **\$this** is used to refer to the object that contains the

operation. The W function application automatically adds the variable $\$this$ as formal parameter of the corresponding rule declaration⁸. Moreover, for simplicity (although these concepts can be handled in ASMs) we assume that an operation cannot raise exceptions (i.e. the set of types provided by the association end `raisedException` is empty) and does not specify constraints.

The W function provides semantics to the **Property** element (an attribute or an association end) of the UML⁺ language by associating it to the corresponding **Function** element of the AsmM involved in the join point definition.

Due to UML and AsmM types identification, as explained in Table 1, the domain of the ASM function denoting a property has to be intended as the ASM domain (usually an **AbstractTD** type-domain) induced from the exposing class of the property, and the codomain as induced from the property's type.

Opaque expressions are straightforwardly matched (see Table 1) into ASM terms.

Remark. Currently, we apply some restrictions to the UML metamodel⁹. We assume exceptions cannot arise from operations, no default values can be specified for the operations parameters, no pre/post conditions and body conditions for operations, no qualifiers (like derived unions and subsetting) as optional part of association ends can be specified, no visibility kinds, only simple classes are treated (i.e. no composite classes with parts and ports), association classes are not yet supported, and only binary associations with none aggregation type are currently permitted. Moreover, as we concerned to PIM level, we assume that class features (both properties and operations) are not *static*, since for static features two alternative semantics are recognized in UML¹⁰ leading therefore to alternative implementations that should instead be taken at PSM level.

5.2 UML⁺ Action Semantics

According to the ASM semantic domain, operations can be invoked on an object (an element of an ASM domain) of a terminal model, given a particular set of substitutions for the parameters of the operation. An operation invocation may cause changes to the values of the properties of that object, or of other objects that can be navigated to, directly or indirectly, from the object's context on which the operation is invoked, to its output parameters, to objects navigable from its parameters, or to other objects in the scope of the operation's execution. An operation invocation may return a value as a result, and in this case the semantics is that of a Turbo ASM rule with return value. Operation invocations may also cause the creation and deletion of objects by executing *extend* ASM

⁸ Since operations are intended as ASM transition rules, within the body of an operation op , if f is a property (an attribute or an association end) in the same object, then $f(\$this)$ must be used as a full name for that property. If $anotherOp$ is another operation in the same object, then $anotherOp(\$this, \dots)$ must be used to invoke that operation.

⁹ They do not limit our approach and can be considered in the future.

¹⁰ A static feature may have different values for different featuring classifiers, or the same value for all featuring classifiers.

rules and update rules of set-functions. Expression evaluations are supported as well.

In addition to these basic actions (property getting/setting, expression evaluations, operation invocation, object creation/deletion), the weaving between the UML and the AsmM metamodels allows us to use more sophisticated ASM rule constructors to express behavior: if-then-else, parallel execution (*par* rules), sequential execution (*seq* rules), finite iteration submachines (*iterate* and *while* rules), non-determinism (*choose* rule), etc., as formally defined in [6].

It is possible to make use of the parallel ASM execution model that (a) easens specification of macro steps (refinement and modularization), (b) avoids unnecessary sequentialization of independent actions, (c) easens parallel/distributed implementations [6]. Furthermore, one can exploit the ASMs feature of incorporating non-atomic structuring concepts (by the constructor *seq* for sequentialization) and finite iteration submachines with return values, exception handling, local values, etc., as standard refinements into synchronous parallel ASMs.

The idea here is to extend the UML class model to allow the definition of ASM transition rules working as “pseudo-code over classes” as scheme of a generic control machine, and allow therefore different granularities of computation step for validating objects behavior, even for parallel/distributed behavioral facets, at PIM level.

6 Implementation

We have been implementing an Eclipse-based integrated environment made of: a UML modeler; the ASMETA/AsmetaS tool, as execution environment; and the AMW (ATLAS Model Weaver) [1] and ATL (ATLAS Transformation Language) [22] to handle the merging process between the UML and the AsmM. The tool implemented at the moment is still a prototype; e.g., we have been carrying our experiments with the EMF-based implementation of the UML 2.x metamodel for Eclipse rather than using directly an external UML visual modeling tool.

Once the ASM semantic model is obtained from a UML⁺ terminal model (see the building function M in Sect. 5.1), several reasoning activities can be carried out, early at PIM level, by exploiting the ASMETA toolset. Model validation is possible by random, interactive, and scenario-based simulation, or by automatic test case generation. Model verification can be done through model checking techniques. Furthermore, this high level ASM model can be exploited for conformance analysis of refined PSMs and code models.

7 Case study: Invoice Order System

The Invoice Order System (IOS), taken from [20], is used as an example to illustrate our approach. The subject is to invoice orders (R0.1). To invoice is to change the state of an order from *pending* to *invoiced* (R0.2). On an order, we have one and only one reference to an ordered product of a certain quantity; the quantity can be different from other orders (R0.3). The same reference can be

<i>UML⁺</i>	AsmM
An active class C	An ASM containing in its signature a domain C as subset of the predefined domain Agent
A non-abstract class C	A dynamic AbstractTD domain C
An abstract class C	A static AbstractTD domain C
An Enumeration	An EnumTD domain
A primitive type	A basic type domain
Boolean	BooleanDomain
String	StringDomain
Integer	IntegerDomain
UnlimitedNatural	NaturalDomain
A <i>generalization</i> between a child class C_1 and a parent class C_2	A ConcreteDomain C_1 subset of the corresponding domain C_2
An attribute a of a class C , type T , kind A function $a : C \rightarrow T$ of kind k , and multiplicity 1	A function $a : C \rightarrow T^*$ of kind k , where T^* is the domain of all finite sequences over T (SequenceDomain)
An attribute a of a class C , type T , multiplicity > 1 , and <i>ordered</i>	A function $a : C \rightarrow \mathcal{P}(T)$ of kind k , where $\mathcal{P}(T)$ is the mathematical powerset of T (PowersetDomain)
An attribute a of a class C , type T , kind A function $a : C \rightarrow B(T)$ of kind k , multiplicity > 1 , <i>unordered</i> and <i>not unique</i>	A function $a : C \rightarrow B(T)$ of kind k , where $B(T)$ is the domain of all finite bags over T (BagDomain)
A navigable association end	See attribute
An operation op of a class C , rule body R , arity n , and owned parameters $x_i : D_i$ ($D_1, \dots, D_n = R$) of rule body R , arity $n+1$, and formal parameters $\$this$ in C and x_i in D_i	The (unique) main rule declaration of form main rule $op = \text{for all } \$this \text{ in } C \text{ do } R$
An opaque expression	A term
OCL constraints	Axioms (optional)
OCL operations/queries	Static functions (optional)

Table 1. W mapping: from *UML⁺* to AsmM

ordered on several different orders (R0.4). The state of the order will be changed to *invoiced* if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product (R0.5). From the set of requirements presented in [20], we focus here on the *Case 1*, which is specified as follows:

R1.1 All the ordered references are in stock.

R1.2 The stock or the set of the orders may vary due to the entry of new orders or canceled orders, or due to having a new entry of quantities of products in stock at the warehouse. But we do not have to take these entries into account.

R1.3 This means that you will not receive two entry ows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state¹¹.

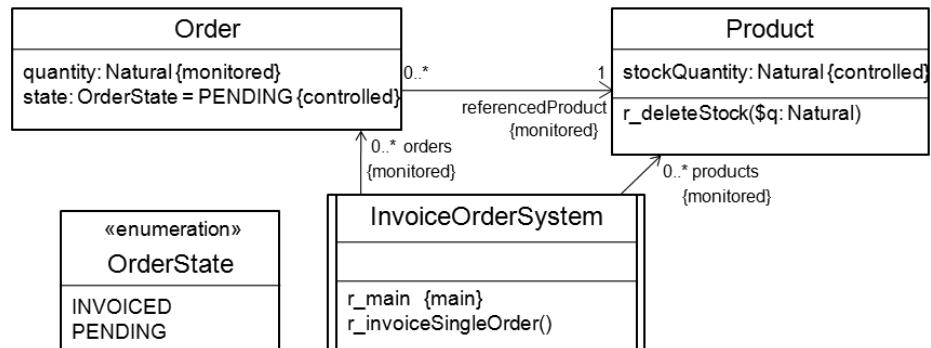


Fig. 4. IOS class model

The UML class diagram in Fig. 4 shows the implementation classes for the Invoice Order System at the PIM level. It shows the internal structure of the system for order management with the essential details at this stage. From a structural point of view, there is: a set of orders (the **Order** class), a set of products (the **Product** class), and a container class **InvoiceOrderSystem** used as *active*¹² class to start the application and thus to invoice orders depending on various strategies. Every order has a state, which can be *invoiced* or *pending*. All the orders are initially pending. Every order refers to a product for a certain quantity (greater than zero) and these data cannot be changed (quantity is a **monitored** property). The same product can be referenced by several different orders. Every product is in the stock in different quantity. The quantity of a product in the stock is only updated by the system (hence it is a **controlled** property) when it invoices some orders.

¹¹ You do not have to take into account the entry of new orders, cancellation of orders, and entries of quantities in the stock. These are subjects for Case 2.

¹² Each instance of an active class has its own thread of control and possibly may coordinate other behaviors.

The behavior of each class is specified by means of ASM transition rules, as shown in the operation compartment of the UML classes. Their definition is reported in Listing 1.1 using the ASMETA/AsmetaL textual notation. The system is intended as a single-agent machine. To invoice orders, the system may follow different strategies. We choose here that of invoicing an order at a time. By invoking the `r_invoiceSingleOrder` operation, the system selects (non-deterministically) on `order`¹³ within a set of orders that are pending and refer to a product in the stock in enough quantity, and simultaneously changes the state of the selected order from pending to invoiced and updates the stock by subtracting the total product quantity in the order to invoice (by the `r_deleteStock` operation). The system keeps to invoice orders as long as there are orders which can be invoiced. The system guarantees that the state of an order is always defined and the stock quantity is always greater than or equal to zero. Note that, non-determinism (`choose` rule) is a convenient way to abstract from details of scheduling. Indeed, in the modeled strategy, per step at most one order is invoiced, with an unspecified schedule (not taking into account any arrival time of orders) and with a deletion function under the assumption that `stockQuantity` is updated only by invoicing.

Listing 1.1. IOS behaviour: single-order strategy

```

rule r.invoiceSingleOrder =
  choose $order in self.orders with orderState($order) = PENDING
    and quantity($order) <=
      stockQuantity(referencedProduct($order))
  do par
    state($order) := INVOICED
    r.deleteStock[referencedProduct($order),orderQuantity($order)]
  endpar

rule r.deleteStock($p in Product, $q in Natural) =
  stockQuantity($p):= stockQuantity($p) - $q

main rule r_Main = r.invoiceSingleOrder[]

```

Many other strategies and particular scheduling algorithms can be defined and refined as well, in order to get a sufficiently precise, complete and minimal, executable PIM model which can serve as a basis for the implementation of various PSMs. Appendix A reports the complete AsmetaL specification associated to the IOS class model in Fig. 4.

8 Conclusion and Future Work

In this paper, we proposed a PIM level behavioral language for structural models based on the ASMs formalism. We focused on an *intra-object* perspective by addressing the behavior occurring within structural entities (like UML class

¹³ Note that a variable `v` is expressed in AsmetaL as `$v`.

models). In the future, we propose to extend the behavioral formalism for the *inter-object* behavior, which deals with how structural entities communicate with each other. The objective of this further effort is to show the applicability of the proposed approach in the area of communication protocols and of inter-process interaction models. This will require identifying suitable join points between structural diagrams describing the collaborative structure of the interactive entities and the AsmM subpart concerning transition rules.

We will also continue working on the implementation of the model execution environment (a virtual machine) and a user interface providing support for a user-friendly model simulation. We could also experiment with a much more lightweight extension of the UML metamodel based on the UML *profile mechanism*, but we preferred a more general matching approach in order to make it reusable for different metamodels rather than only for UML-based metamodels.

Moreover, we want to connect PIM executable models with PSM executable models written with the SystemC UML profile in the context of a model-based development process for embedded systems and System-on-Chip (SoC) [34].

References

1. The AMW (ATLAS Model Weaver website). <http://www.eclipse.org/gmt/amw/>, 2007.
2. The Abstract State Machine Metamodel website. <http://asmeta.sf.net/>, 2006.
3. J. Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.
4. E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2005.
5. E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In Y. G. et al., editor, *Abstract State Machines. Theory and Applications*, volume 1912 of *LNCS 1912*, pages 223–241. Springer, 2000.
6. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
7. A. Cavarra, E. Riccobene, and P. Scandurra. Mapping uml into abstract state machines: a framework to simulate uml models. *J. Studia Informatica Universalis*, 3(3):367–398, 2004.
8. K. Chen, J. Sztipanovits, and S. Neema. Compositional specification of behavioral semantics. In *DATE*, pages 906–911, 2007.
9. B. Combemale, P. G. X. Crégut, and X. Thirion. Towards a formal verification of process models's properties - simplepd and tool case study. In *9th International Conference on Enterprise Information Systems (ICEIS)*, 2007.
10. K. Compton, J. Huggins, and W. Shen. A semantic model for the state machine in the Unified Modeling Language. In Proc. of Dynamic Behavior in UML Models: Semantic Questions, UML 2000, 2000.
11. G. Csertán, G. Huszér, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *ASE*, pages 267–270. IEEE Computer Society, 2002.

12. D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report 06.02, LINA, 2006.
13. Eclipse Modeling Framework. <http://www.eclipse.org/emf/>, 2008.
14. OMG. Semantics of a Foundational Subset for Executable UML Models, version 1.0 - Beta 1, ptc/2008-11-03, 2008.
15. A. Gargantini, E. Riccobene, and P. Scandurra. Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, DTI Dept., University of Milan, 2006.
16. A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based simulator for ASMs. In A. Prinz, editor, *Proceedings of the 14th International ASM Workshop*, 2007.
17. A. Gargantini, E. Riccobene, and P. Scandurra. Ten reasons to metamodel ASMs. In *Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis, LNCS Festschrift*. Springer, 2007.
18. A. Gargantini, E. Riccobene, and P. Scandurra. A semantic framework for metamodel-based languages. *Journal of Automated Software Engineering*, 2009, in print.
19. The Generic Modeling Environment (GME). <http://www.isis.vanderbilt.edu/Projects/gme>, 2006.
20. H. Habrias and M. Frappier. *Software Specification Methods: An Overview Using a Case Study*. Wiley, 2006.
21. D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.
22. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.
23. J. Jürjens. A UML statecharts semantics with message-passing. In *Proc. of the 2002 ACM symposium on Applied computing*, pages 1009–1013. ACM Press, 2002.
24. I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based dsl frameworks. In *OOPSLA Companion*, pages 602–616, 2006.
25. OMG. The Model Driven Architecture (MDA Guide V1.0.1). <http://www.omg.org/mda/>, 2003.
26. H. Miao, L. Liu, and L. Li. Formalizing uml models with object-z. In *ICFEM '02: Proc. of the 4th Int. Conference on Formal Engineering Methods*, pages 523–534, London, UK, 2002. Springer-Verlag.
27. P. Mohagheghi and V. Dehlen. Where is the proof? - a review of experiences from applying mde in industry. In I. Schieferdecker and A. Hartman, editors, *ECMDA-FA*, volume 5095 of *LNCS*, pages 432–443. Springer, 2008.
28. A. M. Mostafa, M. A. Ismail, H. E. Bolok, and E. M. Saad. Toward a Formalization of UML2.0 Metamodel using Z Specifications. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNP 2007. Eighth ACIS International Conference on*, volume 1, pages 694–701, 2007.
29. P.-A. Muller, F. Fleurey, and J.-M. Jezequel. Weaving Executability into Object-Oriented Meta-Languages. In *Proc. of ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.
30. I. Ober. More meaningful UML Models. In *TOOLS - 37 Pacific 2000*. IEEE, 2000.
31. OMG. Object Constraint Language (OCL), v2.0 formal/2006-05-01, 2006.
32. C. Raistrick, P. Francis, and J. Wright. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

33. E. Riccobene and P. Scandurra. Towards an Interchange Language for ASMs. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines. Advances in Theory and Practice*, LNCS 3052, pages 111 – 126. Springer, 2004.
34. E. Riccobene and P. Scandurra. Model transformations in the UPES/UPSoC development process for embedded systems. *Innovations in Systems and Software Engineering*, 5(1):35–47, 2009.
35. M. Scheidgen and J. Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA-FA*. Springer, 2007. LNCS.
36. C. Snook and M. Butler. Uml-b: Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.
37. OMG. UML v2.2 Superstructure, formal/09-02-02, 2009.
38. The Xactium XMF Mosaic. www.modelbased.net/www.xactium.com/, 2007.

A ASM model for the IOS system

Listing 1.2. ASM model for the IOS (single-order strategy)

```

asm InvoiceOrderSystem //Case 1
import STDL/StandardLibrary
signature:

//Domain declarations
domain InvoiceOrderSystem subsetof Agent
abstract domain Order
abstract domain Product
enum domain OrderState = { INVOICED | PENDING }

//Function declarations
dynamic controlled state: Order -> OrderState
    //the product referenced in an order
dynamic monitored quantity: Order -> Natural
    //the quantity in the order
dynamic controlled stockQuantity: Product -> Natural
    //the quantity in the stock
dynamic monitored referencedProduct: Order -> Product
    //the product referenced in an order
dynamic monitored orders: InvoiceOrderSystem -> Powerset(Order)
    //the referenced orders
dynamic monitored products: InvoiceOrderSystem -> Powerset(Product)
    //the referenced products
definitions: /*----- Rules for case 1 (single-order strategy) -----*/
macro rule r_deleteStock($p in Product ,$q in Natural)=
    stockQuantity($p):= stockQuantity($p) - $q

rule r_invoiceSingleOrder =
    choose $order in self.orders with orderState($order) = PENDING
        and quantity($order) <= stockQuantity(referencedProduct($order))
    do par
        state($order) := INVOICED
        r_deleteStock[referencedProduct($order),orderQuantity($order)]
    endpar

/*----- main rule -----*/
main rule r_main =
    r_invoiceSingleOrder[]

//A possible initial state
default init s_1:
    function state($o in Order) = PENDING
        //default value from the class diagram
    function stockQuantity($p in Product) = 100n

```

Behaviour Modelling Notation for Information System Design

Audris Kalnins, Edgars Celms, Elina Kalnina, Agris Sostaks

University of Latvia, IMCS, Raina bulvaris 29, LV-1459 Riga, Latvia
audris.kalnins@lumii.lv, edgars.celms@lumii.lv, Elina.Kalnina@lumii.lv,
agris.sostaks@lumii.lv

Abstract. Problems related to behaviour modelling within the platform independent model (PIM) during the model driven design are discussed in the paper. The emphasis is on design problems for information systems, especially on building a behaviour draft. At first issues in the traditional approach using sequence diagrams are discussed. Then a new approach based on activity diagrams is proposed. An extension of activity diagram notation specifically oriented towards comprehensive and readable behaviour design description is presented.

1 . Introduction

The Usage of UML for model driven software development (MDSD) has become an everyday practice for many software developers. Frequently the approach proposed by early MDA guidelines [1] by OMG is observed to a degree, and platform independent (PIM) and platform specific (PSM) models are built. Both pure UML [2] and various its profiles are used as the modelling notation.

Alternatively, many MDSD approaches are based on domain specific languages (DSL). However, here true success stories are typically related to specific domains [3] where an adequate DSL serves both as the modelling and development language being compiled directly to executable code. In this paper we do not cover the DSL approach. Instead, we are interested in general software development where UML is still dominant. More specifically, our main domain of interest is the development of “general purpose” information systems.

The static structure of software is adequately described by UML class diagram notation. The UML profile mechanism serves well for some missing elements. At the PIM level most frequently the basic class notation is sufficient. This notation is adequately supported by most of UML tools, including some built in model transformations from PIM to PSM and PSM to code.

The situation with behaviour modelling is not so bright. UML has several diagram types for behaviour description: sequence, activity, state and some combinations of them. None of them is universal, each type is best for some specific purpose.

Sequence diagrams are the most used notation for general class interaction description. In this paper the authors are particularly interested in just this application. To be more precise, we mean the step of PIM building frequently named “use case

realization” [4, 5]. On the basis of requirements (which typically are organized into use cases) in this step the first draft of system behaviour is built. This behaviour draft includes finding of basic class operations and definition of class interactions, including the exchanged data and the execution sequence. This kind of behaviour design within a model driven development is based on sequence diagrams, as a rule [4,5]. Typically, the goal is to realize all scenarios related to the given use case, therefore several sequence diagrams may be built. This step may also include automatic generation of draft behaviour from more formalized requirements.

However, the usage of sequence diagrams for the given goal has a lot of unsolved issues. They are sufficient if we want to present the bare invocation chain between class operations. But as soon as we want to represent the control structure in a more detailed way some inconveniences appear. Things are even worse if we want to represent also the basic data flow between operations in the chain. The introduction of UML 2 has made some problems more acute, though some improvements are present also.

In this paper we analyze the problems appearing when sequence diagrams are used for general software behaviour design. This is done on the basis of a simple but typical example, related to building a PIM for an information system. Some proposals for improving UML sequence notation are also given.

Other behaviour diagrams are typically used for more specific purposes. UML state diagrams (statecharts) are typically used in software development for embedded systems which frequently are state based by the very nature. Frequently profiles or DSLs based on UML statechart elements serve as direct development languages for such systems [3]. We will go no deeper in this direction.

Activity diagrams in UML have a more general role. They incorporate also the basic actions package. This package in fact is a sort of simple programming language without a concrete syntax. Currently OMG has an ongoing effort for defining precise executable subset of UML based on activity diagrams and actions [6]. In addition, there is an intention to provide a usable textual syntax for the action sublanguage [7]. The goal of all these activities evidently is to provide a usable DSL directly within UML for the development of some kind of systems. Most probably, the emphasis again will be on embedded systems with all the issues of concurrency and so on. Another typical application of activity diagrams is for workflow design [8], where they have to compete with the popular BPMN notation [9]. Typically, here profiles are required too [10].

In this paper we briefly present a new extension of activity diagrams specifically oriented towards behaviour design description. Our goal is to have a readable behaviour notation which would nevertheless cover constructs typically used in behaviour design at the PIM level (in the same “use case realization” step). The positive aspects of sequence notation are preserved as far as possible, but in the new notation all relevant aspects beyond a pure invocation chain can also be represented adequately. Certainly, some textual notation for a small subset of actions has to be provided too. In order to preserve the notation simplicity we aim at general purpose systems (information systems and similar ones) where thorny issues such as concurrency are not so important. Another objective for the proposed notation is that it should be “model transformation friendly” – both for generation and for

transformation to PSM notation. It should be noted, that this notation can be treated as a kind of class diagram extension too.

The section 2 of this paper is devoted to problems appearing when sequence diagrams are used for behaviour design and some proposals for improvement. The proposed new activity notation for behaviour design is presented in section 3.

2 . Sequence Diagrams for Behaviour Design

Most of MDSD methodologies [4,5] propose to use sequence diagrams for representing class behaviour design decisions. Classes required for accomplishing a task are shown as lifelines. The design process is started from the main “external” operation triggering the whole given task (typically, the given use case). Looking at the current operation invocation the designer tries to split the current task into smaller subtasks. They are forwarded to some other classes as operation invocations within the body of the current operation under design. If the other class already has the required operation it is invoked (an operation invocation message added to the sequence diagram). Otherwise at first the required operation is defined and added to the relevant class, then the invocation is added. Thus in fact a co-design of class and sequence diagrams occurs. Frequently a set of design patterns is used in this process (see e.g. [4]). A similar approach sometimes can be used for automated design when the PIM model can be partially built by model transformations from a set of formalized requirements [11].

The sequence diagram notation is perfect as long as we want to store nothing more than the relevant invocation/return chain for one case (“scenario”). And for messages corresponding to operations only the signature is recorded. This design level is equally well supported by UML from 1.4 [12] to 2.2 [2].

However, in many cases the designer wants to store more information than just “naked” invocation chain. On the one hand, there is a desire to define some control structure within the operation body under design. Branching is required to show several alternative scenarios (of a use case) within one diagram. Certainly, an alternative is to build a separate diagram for each scenario. But it is even more important to define some loops for performing iterative actions for a whole collection of related data. On the other hand, typically there is a great desire to represent some data flow within the operation body under the design – what class attributes and local variables are used as operation arguments, where the returned data are stored and so on. At least typical textbooks [4,5] use this approach as a rule.

That is where the problems with sequence diagrams start. UML 2 offers a fragment notation for defining a branching in diagram. This notation is formally complete (in the sense that any structured branching can be defined). Though, sometimes this notation looks quite lengthy in practice. The possibility to use sequence subdiagrams (via interaction use fragment) in UML 2 may help a little. However, the provided facilities for loops are fairly incomplete – only a simple WHILE-loop can be defined more or less adequately. The situation with data flow representation in sequence diagrams is even more complicated.

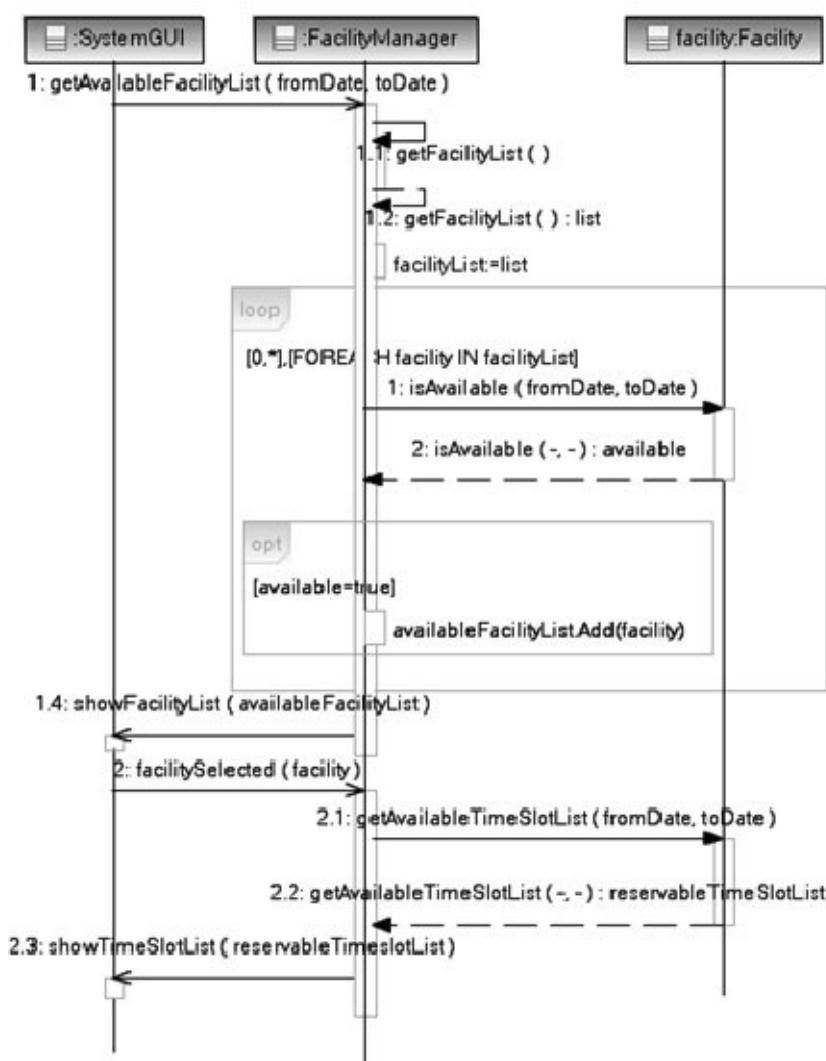


Fig. 1. Sequence diagram example.

In order to explain the situation in more details let us have a small example. Let us imagine that we are building an information system for a Fitness Club. This club has a number of Facilities which are used by customers for exercises. Customers can book for certain regular Time slots for these Facilities (e.g., Monday to Friday from 7:30 to 8:00). We design a small fragment of the use case Reserve Facility, more precisely, the initial part of the scenario where the customer gets the list of Facilities available for the given time period. Then the customer can select a Facility and see the list of Time slots available for this facility. Fig. 1 shows the sequence diagram which has

been built during this design process and Fig. 2 the corresponding fragment of the class diagram, where all required operations have been incorporated. These diagrams have been built using the RSA tool [13] from IBM, version 7.5.1. The notation corresponds to UML version 2.1 supported by this tool.

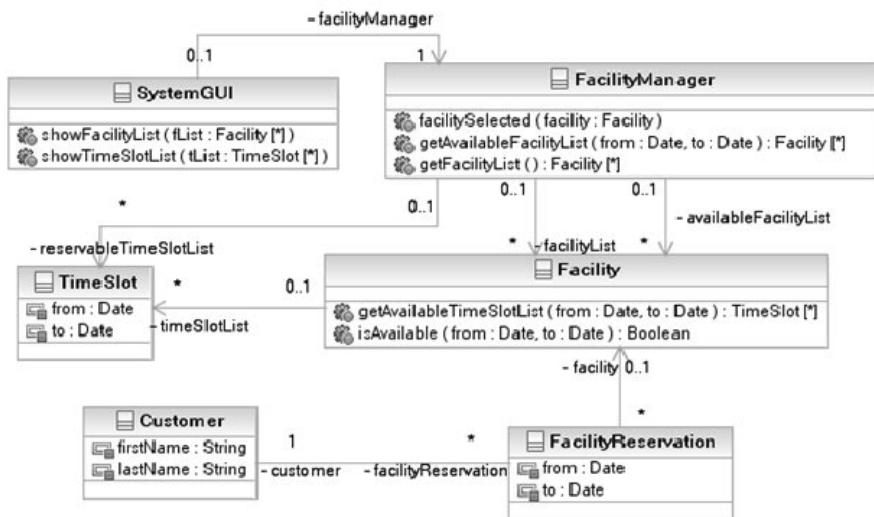


Fig. 2. Class diagram fragment.

We assume here that the Façade Controller pattern [4] was used in the design process. The user interaction was built according to the general MVC pattern. We repeat once more that the PIM level is assumed so no specific GUI framework is chosen and so on. Due to all this we represent the whole GUI layer by one “abstract” GUI controller class *SystemGUI*. The class *FacilityManager* plays the role of Façade Controller for this use case. The sole domain-specific class visible in the sequence fragment is *Facility*. According to general principles frequently used in MVC, all invocations from/to GUI level are asynchronous (more precisely, without return), but all other ones are synchronous (with return).

Now some sequence notation issues can be explained in detail. We have decided that the execution of `getAvailableFacilityList` starts with a call to `getFacilityList` operation within the same class. The returned complete Facility list should be stored by means of the navigable association *facilityList* (which has * multiplicity). In UML 1.4 this could be specified as a single message

facilityList = getFacilityList()

with a solid arrowhead (the return message is implicit). In UML 2, the return message is required if you want to specify some return value (this value is permitted only on return messages). The second `getFacilityList` message in Fig. 1 is this return message, returning the value list. The standard (UML 2.1, 2.2) asserts that an assignment of the

kind *facilityList* = ... is also permitted on the return message. However, it asserts further that such a notation means the return message combined with a separate assignment action execution. Thus, these two separate elements should be created in the abstract syntax (UML domain) model. Certainly, no tool builder (including IBM Rational) is eager to provide such a complicated facility. Therefore an explicit assignment action has to be added to the invocation and return (unfortunately, action execution is not fully implemented in RSA 7.5 sequence diagrams, so a little forgery is done here). To sum up, the only legal way is to use the clumsy notation you see in the upper part of Fig. 1 (two messages and an assignment action). A similar comment applies to all other places where return values have to be processed further. A possible remedy could be quite simple – extend the metamodel and constraints so that the “old” notation becomes legal.

Another issue is related to loops. The returned complete facility list has to be filtered – only those facilities should be left which have unfilled time slots within the specified period. So we decide to have a loop iterating over the list and invoking the *isAvailable* operation (in the Facility class) for each list element. The elements returning true are stored in a new list (held by the *availableFacilityList* association). In any up-to-date programming language (Java, C#, ...) this could be described by a simple iterator-style (*for*) loop. Therefore it is natural to expect the same level of abstraction in UML too. Unfortunately, UML 2.2 loop offers only the minimum/maximum bounds (nothing better than the default 0 and * normally fit here) plus a Boolean guard expression (obviously, denoting the while-condition). We propose to use some explicit iterator over a collection (e.g., *FOREACH facility IN facilityList*). The iterator variable (*facility*) has a type (here, *Facility*) to be simply implied from the collection itself. The name of this variable is used as the relevant lifeline name, thus denoting that the current *Facility* instance is always used. The issue of returned Boolean value (*available*) is similar to the case discussed above. Adding to the result list is clearly an explicit action (we have chosen a Java-like *Add* notation). To sum up, our proposal is to add an explicit iterator to loop syntax and permit to drop bounds where senseless (in fact, the metamodel already permits to drop the bounds). It should be noted that a similar proposal has already appeared (see the for-loop in [14]).

To conclude, sequence diagrams could become a usable design notation if the mentioned and possibly some other extensions would be included. It should be noted that extensions of this kind are vital for processing the model by transformations (e.g., to create the initial PSM from PIM). Transformations can do some sensible job only if more than bare invocation chain is specified in an unambiguous way.

An important aspect is the practical necessity to represent a set of related behaviours (e.g., those covering a use case or a group of use cases). If a separate diagram is used per scenario branch, there typically will be many “streamlined” (without alt-fragments) diagrams. Alternatively, a lot of diagrams may be merged using alt-fragments, but these merged diagrams will become quite large. Thus the total size of graphical description of the required system behaviour will be quite large in any case. While this is not particularly important for automated processing (generating from requirements or transforming to PSM) it is critical for manual evaluation and updates. Since such manual activities are a must for real MDSD style development this is one more problematic aspect of sequence notation.

Certainly, principal quality issues arise for a set of sequence diagrams describing the given fragment of system behaviour, more precisely, issues of consistency and completeness. A substantial answer to these issues is far beyond the scope of this paper. But a couple of short comments can be given. In practical terms the description completeness (e.g., whether there are missing else-branches) is not so critical, it simply means the design process has to be continued. The consistency (e.g., the same operation body is described in two sequence diagrams in two conflicting ways) is critical and must be validated. However, it is more about the correctness of requirements from which the behaviour has been obtained. If there are automated transformations building a PSM model from the given PIM including behaviour, it could be their task to check the consistency of operation definition throughout the whole fragment. However, for manual quality evaluation the same readability of the description as a whole is critical.

The final comment on sequence diagrams in UML 2 is more related to tool building. The metamodel package for Interactions is excessively complicated (see similar remarks also in [15]). For example, a simple synchronous operation invocation message (with one parameter and a return value) from class to class requires 18 UML domain instances to be created. Or, to reach the corresponding operation from a message three links must be navigated (but 7 metamodel classes are involved to find this fact). All this significantly hinders the use of UML 2 sequence diagrams in automated MDSD – both to build a sequence diagram by transformations, or to analyze it. May be, all this explains the fact that there was Java code generation from PSM sequence diagrams in UML 1.4 (in Borland Together tool), but no known tool does this for UML 2. The popular “reversing” of sequence diagrams from code does not count – here only the invocation chain is built. We consider the generation of method “body skeletons” from behaviour descriptions in PSM an essential part of MDSD.

3 . Notation for Behaviour Design Based on Activity Diagrams

The arguments in the previous section show sequence diagrams in their current form have significant drawbacks as a behaviour design notation for “use case realization”. Therefore authors have considered also alternative notations for this purpose. One source of inspiration could be the fact that flowcharts have been used for behaviour formalization for more than 40 years and are still used in a semiformal way. This suggests the use of activity diagrams as a kind of flowchart formalization. The current OMG activities related to action languages [6,7] also lead in the same direction. Therefore authors have tried to find a usable behaviour notation based on activities.

The activity notation as it is in UML 2 is not directly well fit for this purpose. The comparison to Java-like action language in [7] shows how clumsy is this notation if followed literally. The main cause again is the extremely low level of some actions and the need to code all data exchange by data flows involving explicit pins. Therefore the authors have tried to find some simple activity extensions which would be sufficient for behaviour design. It should be noted that our goal was not to provide a complete executable language as in [6,7] since the behaviour specification is meant

to be extended manually at the PSM level and at the code level (as the general MDSD approach suggests). Only the essential aspects of behaviour should be definable in a readable way.

In the result a draft proposal for such a notation is given in this section. The main idea is to borrow something from sequence diagrams and incorporate this into activity diagrams. The main desire is to preserve as much as possible the clear visibility of invocation chains which is excellent in sequence notation. Also the visibility of order in which actions occur should be preserved where possible (but this is not possible always).

The basic ideas are the following. Each class participating in a design fragment is given a swimlane. This swimlane contains the operation definitions of this class which are relevant to this fragment. Each operation definition is a simplified nested activity, with title equal to the operation signature. This activity contains actions required for the operation body behaviour description at the desired level of details. A typical action is an operation invocation (of the same or another class). From this action a special kind of arrow is drawn to the corresponding operation definition (with or without return). Other actions are assignment actions and some basic data processing actions we want to show at this level. Contents of all actions are defined in a textual form (a very simple “action language”). The control structure within an operation definition is shown by means typical to activity diagrams. Only control flows are used, but not object flows. Decisions with guards attached to outgoing flows are used for branching. A graphical loop notation is used (this notation in fact is borrowed from an early draft of UML 2, currently there is no specialized symbol for loop, the general structured node must be reused).

Fig. 3 shows the same behaviour fragment from Fig. 1 in our proposed notation. The same class definitions in Fig. 2 are used for reference. Further details of our proposal will be explained on the basis of this example.

The operation definitions within a swimlane (corresponding to a class) are naturally ordered from top to bottom. Though this ordering has no formal meaning it can be used to show a typical execution sequence. It can be seen in Fig. 3 that due to the MVC based design the complete behaviour in fact splits into independent fragments, with each fragment defined in a “sequence style” – an operation invocation, invoked body, nested invocations and their bodies and so on, including also returns. Each fragment starts and ends within the *SystemGUI* swimlane (this is typical for MVC based designs). Since at this abstraction level no more details on GUI can be shown some constraints on execution order remain hidden – for example, a selection in a form can be done only after this form has been shown. In sequence diagram these constraints were implicitly imposed by the vertical ordering of events. In this notation an explicit “temporary dependency” arrow (a dashed arrow) must be used since the vertical ordering has only informal meaning. This special kind of arrows is needed only in GUI-related swimlanes (only one such arrow in Fig. 3).

The execution order within an operation body is specified by control flows. Execution starts from an action without incoming control flow. If required for readability, the activity start symbol (the black dot) can be used, but typically the vertical ordering within the body is sufficient for easy finding the start.

Thus the same vertical ordering of events as in sequence diagrams can be preserved for readability, but a greater flexibility is provided if needed.

An operation invocation is specified in a textual style, with arguments as in sequence diagrams. The invocation without return (a thick filled arrow head at one end) has no more options, but for invocation with return (an open arrow head at the other end too) an assignment can be combined with the invocation (in a syntax similar to the one used in UML 1.4 sequence diagrams). If required, the returned value can be specified on the invoke-return arrow. Explicit assignments are not frequently needed in this notation but can be used if required. Other data related actions (*Add* action in Fig. 3) are defined in the same Java-like style we propose in sequence diagrams. The syntax for loop head is also the same as we propose for sequence diagrams. If required, subdiagram invocation can be used for structuring of the description as in standard activity diagrams.

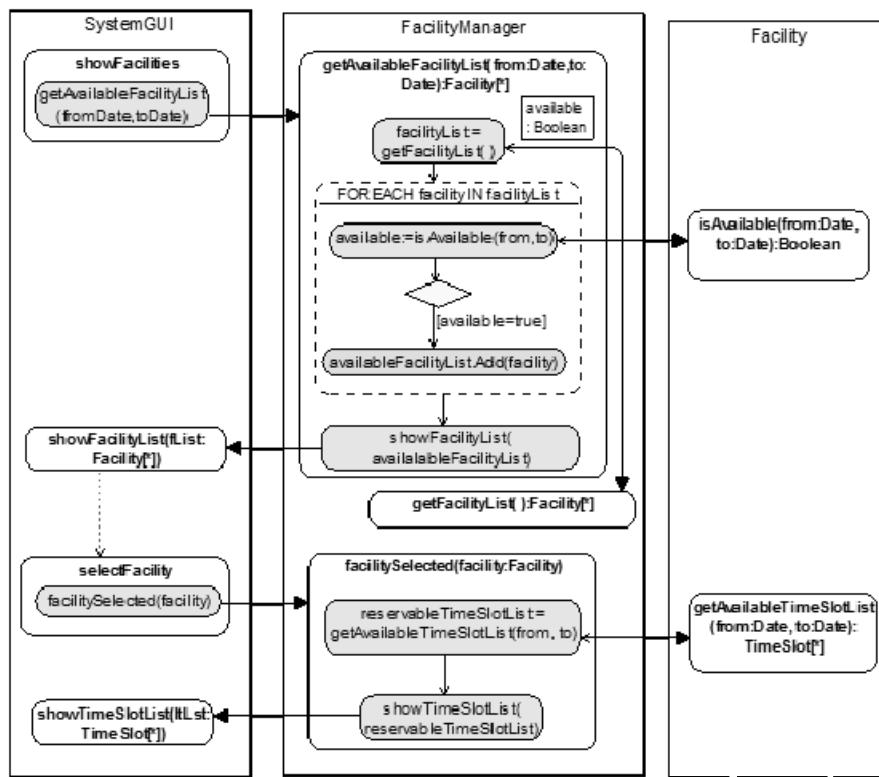


Fig. 3. Example behaviour description in the proposed notation

One more specific facility is the definition of local variables within operation bodies. In contrast to attributes, such variables can not be defined in the class diagram. Certainly, the use of Boolean variables (*available* in Fig. 3) is not very typical at this level of abstraction, but nevertheless it illustrates the intended use of

this construct – to make the data flow completely explicit if required. There is no similar facility in sequence diagrams.

This completes the description of the proposed notation. The example shows that nearly all positive aspects of sequence notation (easy traceability of invocation chains, explicit ordering of events) have been retained, but much of flexibility and precise data flow description is gained. We might call the notation “activity sequence diagrams” (ASD). We remind that the intended application area for the notation is information (and similar) system design. Therefore concurrent execution and similar complicated features currently are not included in the notation in order to keep it simple. The notation has been checked on some larger examples and has revealed no flaws.

It should be noted that we can look at the proposed notation as an extension of class diagram too. Simply, in each class the relevant operations are expanded into subdiagrams (activities) of their own, with invocation arrows between classes visible. This would permit to keep also associations in the same diagram. However, the value of such notation is still to be checked (it may be too overcrowded with various lines).

The same way as for activity diagrams, a set of ASD would be required to define a certain behaviour fragment (a group of related use cases). Since alternative scenarios are combined in one diagram in a more compact way than for sequence notation, the gain in total diagram size would be more significant. This should be crucial for manual behaviour evaluation and development. The semantics of behaviour description in several diagrams is quite obvious. If a class appears in several diagrams the operation definitions are merged. Certainly, definitions of the same operation in several diagrams must be consistent (the same situation as for the sequence notation). In general, a more formal semantics of ASD notation could be provided, but this is not the topic of this paper. The intuitive semantics for all “normal” cases should be clear from the example.

We conclude with some tool related aspects of the notation. Though it is quite similar to UML activity notation, it is formally not a UML profile. A prototype tool for this notation was built by a small extension of an activity diagram editor within a generic metamodel based tool platform [16], Fig. 3 was obtained by this tool. The metamodel extensions were also minor, in addition the mapping between the diagram notation and the domain metamodel is much simpler than for sequence diagrams. This fact makes also the corresponding model transformation development much easier than in the case of sequence notation (both for generation and transformation to PSM). All this shows that a complete MDSD tool support based on this notation would not be very expensive.

4 . Conclusions

In this paper the possible solutions for behaviour description during the MDSD process have been analyzed. It is obvious that some solution for this description is required for true success of standard MDSD approach. The main deficiencies of the currently most used sequence diagram notation are discussed on an example. Several proposals are given how to improve the usability of UML 2 sequence diagrams.

As an alternative a new activity based notation is proposed for the behaviour description. This notation preserves most of the positive aspects of sequence diagram notation and gives a greater flexibility at the same time. Especially, the data flow description facilities are significantly extended. The most significant gain is expected in the readability and manageability of a complete behaviour description for a certain system fragment.

As far as authors know this is the first attempt in this direction (except for some trivial trials to generate code from activity diagrams). Some other proposals either try to formalize more deeply the sequence diagram notation (live sequence charts [17]) or extend activity diagrams with object diagram fragments (early versions of story driven modelling [18]). But none of them has the easy readable behaviour description as the main goal.

The proposed notation still has to be checked on large real life examples. This could be done with a reasonable effort since the first prototype tool for the notation has already been built.

References

1. Object Management Group. MDA Guide. Version 1.0.1., omg/2003-06-01, <http://www.omg.org/docs/omg/03-06-01.pdf>.
2. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.2, formal/09-02-02, <http://www.omg.org/spec/UML/2.2>.
3. Kelly, S., Tolvanen, J-P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, Ltd., 2008.
4. Larman, C. Applying UML and Patterns (3rd edition). Prentice-Hall, 2004.
5. Arlow, J., Neustadt, I. UML 2 and the Unified Process : Practical Object-Oriented Analysis and Design (2nd Edition). Addison-Wesley, 2005.
6. Object Management Group. Semantics of a Foundational Subset for Executable UML Models, ptc/2008-11-03, <http://www.omg.org/spec/FUML/1.0/Beta1>.
7. Object Management Group. Concrete Syntax for a UML Action Language Request For Proposal, ad/2008-09-09, <http://www.omg.org/docs/ad/08-09-09.pdf>.
8. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M., Wohed, P. On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. APCCM '06: Proceedings of the 3rd Asia-Pacific conference, pp. 95-104, 2006.
9. Object Management Group. Business Process Modeling Notation (BPMN). Version 1.2, formal/2009-01-03, <http://www.omg.org/spec/BPMN/1.2>.
10. Kalnins, A. Vitolins, V. Use of UML and Model Transformations for Workflow Process Definitions. Baltic DB&IS'2006, Vilnius, Lithuania, July 3-6, pp. 3-14, 2006.
11. Leal, L., Pires, P., Campos, M. Natural MDA: Controlled Natural Language for Action Specifications on Model Driven Development, OTM 2006, LNCS 4275, pp. 551–568, 2006.
12. Object Management Group. OMG Unified Modeling Language Specification. Version 1.4. September 2001, formal/2001-09-67, <http://www.omg.org/spec/UML/1.4>.
13. IBM Rational Software Architect (RSA) tool. <http://www-01.ibm.com/software/awdtools/architect/swarchitect>.
14. France, R.B. Realizing the MDE Vision, LMO (Langages et Modeles a Objets), Invited talk, Toulouse, France, 2007. <http://www.cs.colostate.edu/~france/Presentations/LMO2007.pdf>.
15. France, R.B., Ghosh, S., Dinh-Trong, T., Solberg, A. Model-Driven Development Using UML 2.0: Promises and Pitfalls. IEEE Computer, vol 39, pp. 59-66, Feb-2006.

- 16.Celms, E., Kalnins, A., Lace, L. Diagram definition facilities based on metamodel mappings. - Proceedings of the 3rd OOPSLA (Workshop on Domain-Specific Modeling) , University of Jyvaskyla, pp.23-32, 2003.
- 17.Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer Verlag, 2003.
- 18.I. Diethelm, L. Geiger, A. Zündorf: Systematic Story Driven Modeling, a case study; Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE 2004, Edinburgh, Scotland, 2004

Composition Semantics for Executable and Evolvable Behavioral Modeling in MDA

Ashley McNeile¹ and Ella Roubtsova²

¹ Metamaxim Ltd, 48 Brunswick Gardens, London W8 4AN, UK

ashley.mcneile@metamaxim.com

² Open University of the Netherlands, Postbus 2960, 6401DL Heerlen,

The Netherlands

ella.roubtsova@ieee.org

Abstract. The vision of MDA is to decouple the way that application systems are defined from the specification of their deployment platform. Achieving this vision requires that Platform Independent models are rich enough to capture the behavior of the application, and to support reasoning and execution of functional behavior.

We focus on state transition modeling as being the best able to support MDA and appraise the two types of state machine (Behavior State Machines and Protocol State Machines) defined in UML. We conclude that, for different reasons, neither has semantics that are well placed to serve as a basis for PIM level behavior modeling.

We propose that state transition modeling can be both simplified and strengthened by providing semantics that support process algebraic composition. We claim a number of important advantages for this. Firstly, it provides a common language for defining a range of behavioral abstractions, including software components, behavioral contracts and cross-cutting aspects. Secondly that it better supports analysis of models, by exploiting the formal analysis techniques of process algebra. Thirdly, the semantics enable model execution and testing at the platform independent level across a wider domain than is possible with current UML formalisms.

1 Introduction

The Model Driven Architecture (MDA), an initiative launched in 2001 by the OMG, aims to promote modeling to a central role in the development and management of application systems. In particular, it suggests that “fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution and further enable interoperability” [14] and source code for a specific platform would be largely or completely generated from the model, thus removing the current expensive coupling between applications and the technologies required to run them.

Moreover, key architects of the MDA vision talk of the need to be able to execute and test an MDA model. Richard Soley, CEO of OMG, says that

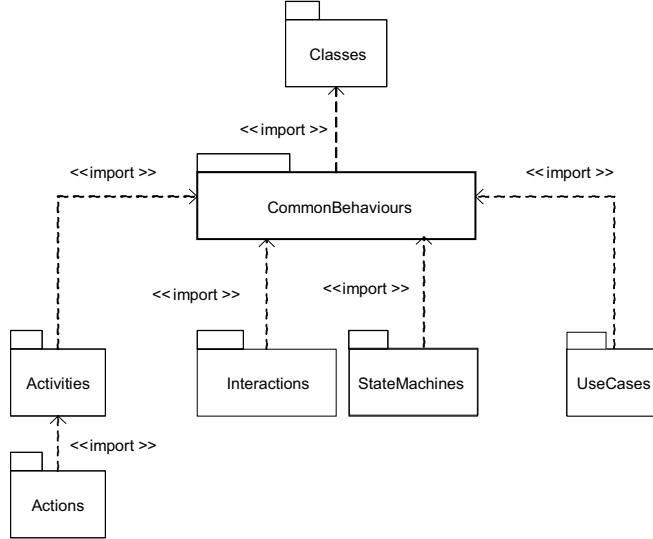


Fig. 1. UML Behavior Diagrams

one of the aims of MDA is that “Models are testable and simulatable” [17]. Oliver Sims, a member of various OMG Task Forces who served for several years on the OMG Architecture Board, says that “The aim [of MDA] is to build computationally complete PIMs” [13]. As Oliver Sims points out, the term *computationally complete* means capable of execution. Executability of a model, whether by interpretation or code generation, is only possible if behavior is fully represented, and the capabilities and properties of the techniques available for modeling behavior are therefore key to achieving the MDA vision.

While there is a general agreement in the MDA community that behavior modeling is essential to the MDA mission and the models should be executable, the behavior modeling notations of UML (Figure 1) [15] do not lend themselves well to executable modeling or model level reasoning.

- *Use Cases* are used to describe scenarios, or episodes, of use of a system by specifying the set of interactions between the system and domain in which it is embedded. Use Cases are described using a combination of natural language and informal diagrams. While “animation” (like playing a movie) of a Use Case may be possible, “execution” in the sense of interactive behavior is not. Their informal and partial nature makes formal reasoning with Use Cases hard.
- *Interaction Diagrams* (Sequence Diagrams and Communication Diagrams) can be used to express interaction of lifelines communication classifiers. Normally an Interaction Diagram presents one scenario of interaction or, with the use of decision and loop constructs, a limited set of related scenarios. As they are scenario based, Interaction Diagrams can be animated to play

out a scenario but, because they are not exhaustive of all behavior, they cannot form a basis for model execution. Some researchers have suggested approaches to composition in order combine scenarios to enable reasoning about the total behavior represented, for example [7, 11], but no composition of Interaction Diagrams is defined in UML.

- *State Machines* exist in two variants: *Behavioral State Machines* (BSM) and *Protocol State Machines* (PSM) [15]. While there is some lack of clarity in their definitions (see, for example, Fecher et al. [9]) state machines do provide complete behavior descriptions and can be used for model based execution. We discuss the UML state machine constructs in some detail in this paper.
- *Activity Diagrams* provide a flow based modeling medium, similar to traditional Petri Nets. They are used to show “the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors” [15]. This means that Activity Diagrams are not suitable for representing the behavior of an object model, which we believe is essential to MDA. While Activity Diagrams can be executed (see, for example, Engels et al. [8]), execution is at the level of a single overall flow and does not encompass the behavior of objects.

This summary suggests that, if we are to be able to capture and represent full and formal descriptions of systems at the model level, State Machines offer the most promising basis. In this paper we look at the forms of State Machine (BSM and PSM) defined in UML and argue that their semantics are not well geared to the aims of MDA. In this paper we suggest that state-transition based behavior modeling can be both simplified and strengthened by introducing support for the parallel composition constructs of process algebras. We explain how this might be done, and argue that this has advantages in terms of improved ability to:

- specify composable behavioral abstractions that support both behavior specification and expression of behavioral contracts,
- abstract on states and actions in a way that supports re-use and the description of cross-cutting behaviors,
- execute models of behavior at the Platform Independent Level.

The remainder of the paper is organized as follows: Section 2 analyzes the semantics of the UML state machine formalisms for MDA purposes, with a focus on their composition semantics. Section 3 contains our proposal for using process algebraic composition semantics in MDA behavior modeling. Section 4 illustrates the ideas with a small example. Section 5 discusses benefits of the proposed ideas and presents some conclusions.

2 State Machine Semantics in UML

The *State Machine Package* in the UML Superstructure document v.2.1 and v2.2 [15] describes a set of concepts that can be used for modeling discrete behavior through finite state transition systems. The State Machine package

defines two behavioral semantics for finite state transition systems: Behavioral State Machines (BSM) and Protocol State Machines (PSM). Our view is that both of these variants are flawed as a basis for MDA development, and in this section we explain this view.

2.1 Behavioral State Machines (BSM)

BSM Semantics A Behavior State Machine usually presents behavior of one classifier. “Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences. During this traversal, the state machine executes a series of activities associated with various elements of the state machine” [15].

A transition label is of the form:

$$\text{event} \text{ [guard]} / \text{action}$$

where *event* specifies the event that triggers the transition, *guard* defines a guard condition that can restrict firing of the transition, and *action* is the action that happens when the transition fires.

The composition semantics for BSMs defines state machines as executing asynchronously and communicating using events, created as a result of actions within the system or in the environment. An event instance is queued until *dispatched*, at which point it is conveyed to one or more BMSs. An event dispatcher mechanism selects and de-queues event instances and an event processor handles the firing of state machine transitions and execution of consequent activity defined by the machines [15].

The consumption of events depends on the active state of a state machine. If an event triggers a transition in the current state of the machine for which it is queued it is dispatched and consumed, and this involves the firing of one or more transitions. If an event can cause two (or more) transitions to fire, which transition is chosen is not defined. If no transition is triggered then either the event is discarded, or it may be held (deferred) for later processing. “A state may specify a set of event types that may be deferred in the state. An event instance that does not trigger any transition in the current state will not be dispatched if its type matches with the type of one of the deferred events. Instead it remains in the event queue while another not deferred message is dispatched instead” [15]. In other words:

- if an event enables a transition in the current state of the machine then it can be dispatched and consumed;
- if an event does not enable a transition in the current state of the machine, but is listed as a deferrable event for this state, it is kept in a queue for later processing;
- otherwise the event is discarded.

The resulting behavior of a population of state machines is, in general, asynchronous and non-deterministic.

BSM Commentary The semantic model used for Behavior State Machine Execution in UML2 (which was first included in UML at version 1.5) is based on the “Recursive Design” method of Shlaer and Mellor [19] whose work has been mainly in the real-time/embedded systems domain. Following the adoption of Shlaer/Mellor semantics into UML the MDA approach based on their ideas has been rebranded as “Executable UML” [18].

The approach is based on using BSMs to model so-called “active objects”: objects whose instances execute autonomously and asynchronously (i.e., as if executing on independent threads) resulting in system behavior that is inherently non-deterministic [20]. It is very hard to reconcile this semantic basis with the characteristics of the business information systems domain, where behavioral issues are related to transactional integrity and business rules, and strictly deterministic behavior of business logic is important to ensure repeatability, auditability and testability. We note that the commercial tools that support Executable UML (such as those from Telelogic, Kennedy Carter and Mentor Graphics) are not well adapted for use in the business information systems domain and are positioned by their vendors to target the real time/embedded market.

The complex composition semantics makes reasoning about behavior difficult. Complete analysis of the behavior of the model must allow, in general, for arbitrary queuing of events between objects and for the accumulation of deferred events. If a model comprises a number of communicating objects this results in a large number of possible execution states for the system as a whole, and reasoning on models is impossible without model checking algorithms. This does not make sense when models are being developed, as they are in most projects, in an iterative manner and subject to frequent change.

While there is some native support in Shlaer/Mellor for behavior abstraction through the use of “polymorphic events”, this has not been included in the UML BSM standard; nor is there is any method to compose multiple machines to form the behavior of a single classifier. This places severe limits on the ability of BSMs to describe generalization/specialization of behaviors or to support behavior re-use. As described in [18], a single object class is modeled with a single state machine, and only concrete classes are modeled. This also means that crosscutting behaviors (aspects) have to be addressed by other means, potentially further complicating model analysis.

2.2 Protocol State Machines

PSM Semantics Protocol State Machine (PSM) are not related to Shlaer/Mellor, and have semantics that are more general and closer to the UML state machine semantics that pertained before the import of Shlaer/Mellor semantics in version 1.5³.

³ It is probable that fracturing of the state machine formalism in UML into two forms was a result of the impossibility of reconciling the semantics of Shlaer/Mellor (reflecting the needs of real-time systems) with the need for a more general capability to specify legal orderings.

PSMs are used to express the legal transitions that a classifier can trigger. A PSM is a way to define a lifecycle for objects, or an order of the invocation of its operations. PSMs can express usage scenarios of classifiers, interfaces, and ports. The effect actions of transitions are not specified in a PSM transition as the trigger itself is the operation. However, pre- and post- conditions are specified, so that the label of a transition is of the form

[pre-condition] event / [post-condition].

The occurrence of an event that a PSM cannot handle is viewed as a precondition violation, but the consequent behavior is left open. “The interpretation of the reception of an event in an unexpected situation (current state, state invariant, and pre-condition) is a *semantic variation point*: the event can be ignored, rejected, or deferred; an exception can be raised; or the application can stop on an error. It corresponds semantically to a pre-condition violation, for which no predefined behavior is defined in UML” [15].

Unlike BSMs, PSMs can (to a limited extent) be composed. “A classifier may have several protocol state machines. This happens frequently, for example, when a class inherits several parent classes having protocol state machine, when the protocols are orthogonal” [15]. In this context, “orthogonal” means that they have a disjoint set of events.

PSM Commentary PSM semantics are simpler and more abstract than the BSM semantics, and this makes them more widely usable and easier to analyze. However, as evidenced by the language used to describe them, PSMs are clearly positioned in UML as *contracts of legal usage*; and this gives it a different meaning and role from that of BSMs. While a contract must specify what is legal, it is **not** concerned with the mechanism by which non-legal behavior is avoided, nor is it required to specify the effect of violation. In other words: A contract cannot be used as the instrument that guarantees its own satisfaction. It would therefore be a logical error to execute PSMs directly or to generate code from them; and other devices must be used in order to ensure that the software that is built complies with the contract PSMs that have been defined for it. To use PSMs as executable models or the basis for code generation in the context of MDA would be inconsistent with this semantic positioning.

3 Our Proposal

Our view is that the state machine formalisms can be both simplified and strengthened by:

- Using a common notational form for both the specification of both contracts and behavior (so eliminating divergence of notation that has emerged in UML between BSMs and PSMs)
- Defining semantics that support process algebraic composition.

The first of these is based on the observation that a machine with behavioral semantics can serve either as a specification or as a contract, depending on the intentions of the author. We contend that there is no penalty, and a good deal to gain, in harmonizing the notations and concepts used across the two.

There is a synergy between these two proposals. In the context of contract definitions it is important to be able to make descriptions that abstract from the full behavior of a classifier, as a contract is normally a **partial** requirement on its behavior. This requirement is met by the second part of the proposal which allows the creation and composition of partial behavioral descriptions.

The composition techniques developed in Process Algebras such as Hoare's CSP [5] and Milner's CCS [16] have so far not made their way into UML, perhaps because the domain of *algebraic processes* (CSP and CCS) and *software models* (UML) have been viewed as too different for the techniques of the former to be used in the latter. However, this is a mistake. Research work into behavior specification techniques, such as those by McNeile et al. [3] and Grieskamp et al. [21], have shown that CSP \parallel composition transplants successfully into software modeling. Other recent work in the context of collaborative service behavior and service choreography specification is making use of CCS and π -calculus, such as the work of Carbone et al. [12]. The proposals we make here exploits and extends the foundations built in this work.

The cornerstone of our proposal is to use a single form of abstract state transition machine, which we call a *protocol machine*, as basis for behavioral modeling. The key property of protocol machines is that their semantics enable composition.

3.1 Definition of a Protocol Machine

A *protocol machine* is, like the state machine constructs of UML, a conceptual behavioral machine. However, unlike the state machine constructs of UML, protocol machines can be composed so that large, complex behaviors can be built by combining smaller, simpler ones.

Protocol machines have the ability to *allow*, *refuse* or *ignore* any action in its alphabet. More specifically, the behavior of a protocol machine is defined as follows:

- It has a defined *alphabet*: a set of actions that it understands.
- In a given state it will:
 - *Ignore* any action that is not in its alphabet;
 - Depending on its state, either *allow* or *refuse* an action that is in its alphabet.
- If it engages in an action it moves to a new state.

The nature of the “actions” in the alphabet of a machine depends on the context and purpose for which the machine has been defined. When defining a single software component, an action represents the receipt of a particular message type from the component’s environment. In the context of message based collaboration in a distributed system, an action represents either the sending or receipt of

a message of a given type. Informally, by *ignoring* an action a machine is saying “I do not know about this action, and have no opinion on whether it can happen or not”, whereas by *refusing* an action a machine is saying “This is an action that I know about (it is in my alphabet) and I know that it cannot happen now”. The distinction between these two, which is not made at all in the semantics of UML state machines, is the basis for parallel composition.

Two further properties are key to the definition of protocol machine behavior:

- If it is starved actions a protocol machine is bound to reach *quiescence*, and only at quiescence is its state well defined. A machine with this property is sometimes called *reactive*. This means that a protocol machine cannot engage in an action that results in a computation that does not terminate.
- A protocol machine is *deterministic*, so the new state it moves to when it allows an action is dependent only on the old state and the action in which it engages.

A protocol machine may, like an object, own attributes; and only the machine that owns an attribute may update it. Updates only take place when a machine allows an action, and constitute part of the change of state of the machine.

3.2 Composition of Protocol Machines

For the purposes of this paper we define two forms of composition, corresponding to CSP \parallel composition and CCS $|$ composition. Other forms of composition are possible but not within the scope of this discussion.

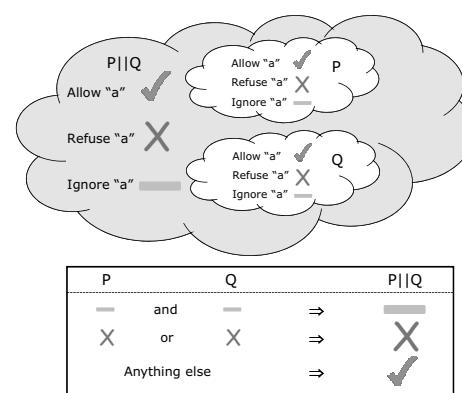
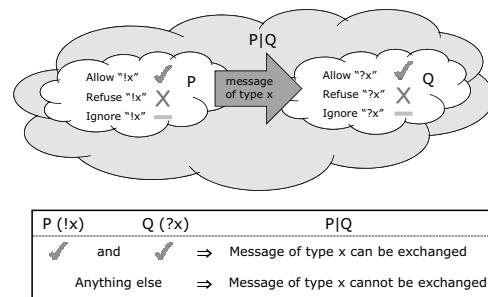
Generally speaking, CSP \parallel is used to compose machines in the formation of a single software component, so the composed parts are within a computing environment that allows them to share state and data; whereas CCS $|$ is used for machines that are distributed in such a way that they cannot share state and data and therefore communicate by exchanging messages.

CSP Composition Suppose two machines P and Q are composed to form $P \parallel Q$. The ability of the composite to engage in an action, a , is defined as follows (see Figure 2):

- If both P and Q ignore a then the composite ignores a .
- If either P or Q refuses a then the composite refuses a .
- Otherwise the composite allows a .

The CSP \parallel composition of two protocol machines is another protocol machine. In particular, it is also deterministic (see [3] for further discussion of this).

Note that this form of composition does **not** have the restriction present in UML for composition of PSMs that the composed machines are “orthogonal”. It is the ability to compose non-orthogonal machines (ones whose alphabets have elements in common) that gives the technique its expressive power.

**Fig. 2.** CSP Composition**Fig. 3.** CCS Composition

CCS Composition Suppose that P is a machine whose alphabet includes a “send action on x ”, represented as $!x$; and Q is a machine whose alphabet includes the corresponding receive action, represented as $?x$. If these machines are composed to form $P|Q$ the behavior of the composite is defined by (see Figure 3):

- If P allows $!x$ and Q allows $?x$ then a reaction on x between them can take place. If the reaction occurs both machines execute their respective actions on x and move to new states.
- Otherwise nothing happens.

The result of CCS composition of two protocol machines is **not** another protocol machine. This is because CCS composition does not, in general, give deterministic behavior of the composite. Suppose P and Q are able to engage in a reaction on y as well as the one on x , then whether the x reaction or the y reaction (or neither) takes place is not determined. We use the term *collaboration* for a set of machines under CCS composition.

3.3 Derived States

Because there is no restriction on how the state of a protocol machine may be determined, we allow machines to have *derived states* as well as the more usual *stored states* driven by the transitions of the state machine. Figure 4 shows an example of a Bank Account described as two machines composed using CSP \parallel . The right hand machine, A_2 , uses derived states (*in credit* and *overdrawn*) calculated on the basis of the *balance* attribute owned and maintained by A_1 .

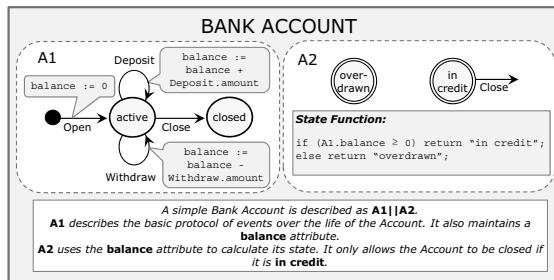


Fig. 4. Bank Account with a Derived State

A derived state is analogous to the familiar concept of a derived (or calculated) attribute, in that its value is calculated “on the fly” by a function when required. The use of derived states is another departure from standard UML state machine formalism, but increases the expressive power to describe action sequencing protocols that depend on the values of stored data. A machine may

use its own attributes and/or those of other, composed, machines to derive its state.

When defining machines, we follow the discipline that a given machine uses either stored states, where updates to the state are defined implicitly by the state-transition topology (as in *A1*); or only uses derived states, where the state values are derived (as in *A2*). This is analogous to the familiar discipline with attributes, where an attribute is either stored or derived. The state icons for derived state machine are given a double outline.

Finally, note that a derived state machine does not have to be “topologically connected”. For instance, the *Close* transition in *A2* does not lead to another state. This is because the state update is not driven by transitions.

3.4 Behavior Re-Use and Aspectual Modeling

The use of composition allows complex behavior to be defined as a composition of smaller, simpler, components. These components can be re-used across the definition of different behavioral entities. Thus, in a banking application that has to support multiple different types of account (Current Account, Savings Account, Student Account, etc.) the basic account behavior described by *A1* could be common to them all and could be composed with machines that represent the particular behavioral rules for each account type. Further discussion of this is given in [3].

Moreover, by exploiting the possibility of defining *generalized states and actions*, this form of re-use can be applied to the definition of crosscutting behavioral aspects. Such generalizations are achieved as follows:

- With the ability to define machines with derived states, we can make one machine generalize over the states of another, rather in the way that the state *in credit* in *A2* generalizes over all non-negative values of *balance* in *A1* (see Figure 4). Such a state, which provides a more abstract view of the states of another machine, is called a *generalized state*.
- If two or more actions are treated identically in the context of a given machine (causing the same transitions and same attribute updates), they can be replaced by a single *generalized action*. This can be thought of as a macro that expands one transition in the machine into a number of transitions with the same start and end states.

A fuller discussion of the use of these techniques in aspectual modeling has been given in [1].

4 Example

We illustrate the idea with a small model of a mobile 'phone equipped with a gaming capability. We use this example to show how a model can be described by using a combination of a *Dynamic View* that shows the behavior of the machines of the model, and a *Static View* that shows how the machines are composed.

4.1 Dynamic View

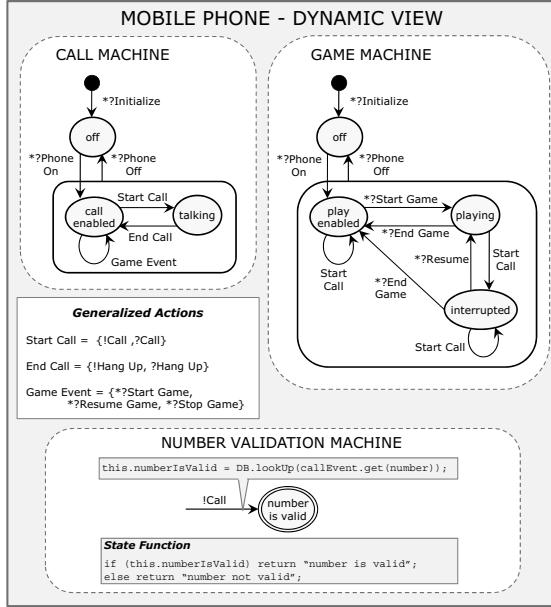


Fig. 5. Mobile Phone - Dynamic View

Figure 5 shows the *Dynamic View*. This shows the state transition representation of the machines of the model. Our model consists of three protocol machines:

Call Machine. This machine handles calls. Once the 'phone is switched on, a call is initiated by *Start Call*. This is a generalized action, representing either *!Call* (this 'phone makes a call) or *?Call* (this 'phone receives a call). Similarly, a call can be ended by either this 'phone hanging up or the remote 'phone hanging up. The generalized action *Game Event* represents events connected with the game facility, and these may only take place when a call is not in progress.

Game Machine. This machine handles game playing. If a call is initiated (by making or receiving a call) and a game is underway, the game is interrupted (the machine moves to the *interrupted* state). It may be resumed later, with the action **?Resume*.

Number Validation Machine. This machine handles validation of called numbers against a list of valid numbers. A call initiation action *!Call* must end in the derived state *number is valid*, and this means that the call event is only allowed if the number is on the list.

Each action in each machine is prefixed by "!" meaning a message sent by the machine or "?" meaning a message or input received by the machine. The "*" prefix is used to signify local events from environment; these do not participate in CCS message reactions.

4.2 Static View

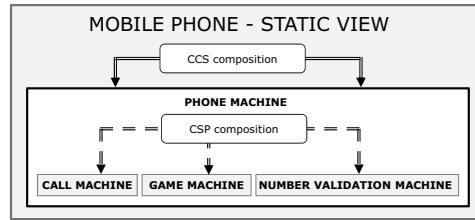


Fig. 6. Mobile Phone - Static View

Figure 6 shows the *Static View* and this shows how the machines of the model are composed. This shows the following:

- *Phone Machine* represents the complete behavior of a 'phone.
- The three machines *Call Machine*, *Game Machine* and *Number Validation machine* are composed using $CSP \parallel$ to form the behavior of a 'phone.
- Multiple 'phones are composed using $CCS |$ to form a collaboration whereby different 'phones engage in calling each other.

The Static View also shows the data attributes:

- In general, every machine owns a set of attributes. For instance the *Call Machine* would own the attribute `my number` being the number of this 'phone.
- Also every message type sent or received will in general carry a set of attributes depending on the message type. Thus a *Call* message will contain the *calling number* of the 'phone initiating the call, and the *called number* of the 'phone being called.

For brevity, we have not shown the attributes of the model in the diagrams.

4.3 Behavior of the Phone Model

As an example of the behavior of the model as described, suppose the 'phone 1234 attempts to call 'phone 4321. This means that 'phone 1234 must allow a $!Call$ action. Consider each of its machines:

- The *Call Machine* allows $!Call$ provided that it is in the state *call enabled*. If the machine is switched off or already on a call, the action is not possible.

- The *Game Machine* allows $\text{!}Call$ provided that it is switched on, as all states have a transition for *Start Call* (which includes $\text{!}Call$). If the machine is in the state *playing*, the $\text{!}Call$ will cause the game to be interrupted.
- The *Number Validation Machine* allows $\text{!}Call$ provided that the action takes it to the state *number is valid*, and this requires that number is on the list of valid numbers. If this is not the case, the $\text{!}Call$ action is not possible.

5 Benefits and Conclusions

In this section, we describe some of the motivating factors for the suggested approach, and give brief conclusions.

5.1 Local Reasoning and Analysis

The CSP \parallel composition technique has the property that it gives *Observational Consistency* (as defined by Ebert and Engels [10]) when applied to protocol machines. This property, as discussed in [1], gives the ability to perform *local reasoning* on models, whereby conclusions about system behavior as a whole can be based on examining single machines in isolation. This is essentially because CSP \parallel composition preserves the *trace behavior* of the composed machines. As has been often noted, for instance by Dantas [6], the ability to perform local reasoning is crucial if intellectual control is to be maintained over a complex model as it grows. As well as this, state-transition models composed using process algebraic techniques can be analyzed using standard model checking algorithms and tools, and this is important in complex cases where “brute force” is needed to check all possibilities (normally where true concurrency is involved).

Together, these provide a basis for ensuring correctness of behavior at the PIM stage of modeling. This, correctly, demotes the role of testing as the means of ensuring that the delivered software works correctly.

5.2 Contracts

Our claim is that protocol machines may be used to describe both *behavior* and *contracts* using a single notational platform. This is a subject of on-going research, but the basis of the idea is simple. Suppose that a protocol machine C is a contract and another machine B is a behavior specification. We say that B satisfies C iff $B \parallel C = B$. This requires that:

- The alphabet of B is a superset of that of C . This is natural, as you would not expect a design to satisfy a contract if it does not recognize all the actions required by the contract.
- By the rules of \parallel composition, an action is only allowed in B if also allowed in C . This means that the states of C can be viewed as defining pre-conditions for the actions of B .

While a full discussion is not possible here, this illustrates the attractive possibility of using a common formalism for both behavior and contractual definitions with a simple formal definition of compliance. This is both more elegant and more powerful than the dual notation approach, with BSMs and PSMs, currently in UML.

5.3 Executability

Protocol models are executable. For example, the ModelScope tool [4] interprets the meta-description of PIM level protocol models. Further discussion of execution of protocol machine models is given in [2].

It is our belief that the formalisms suggested here are more domain neutral than those in the current UML. Whereas the UML BSM semantics has a definite “real time systems” flavor, protocol machines have been used to model database and business process centric systems and have proved to be applicable to these domains.

5.4 Conclusion

Behavior modeling in MDA should be simple, executable and extensible. Our examination of the state transition formalisms of UML suggests that they do not possess these properties. The main contribution of this paper is a proposal to make process algebraic composition techniques central to state transition behavior modeling in MDA. We argue that the capability to compose behavioral models enables behavior re-use and potentially, because they can be used to describe behavioral contracts as well as behavioral specifications, eliminate the need for two forms of state-transition model (BSM and PSM) used by UML.

The composition semantics presented in this paper are based on the *protocol machine* abstraction. Although relatively simple in concept, protocol machines support the modeling of processes and objects that possess and maintain data attributes and, by allowing states to be derived as well as stored, enable the modeling of cross-cutting concerns using state and action abstractions. In addition their simple compositional semantics make them amenable to analysis, both human reason and machine based.

References

1. A. McNeile and E. Roubtsova. CSP parallel composition of aspect models. In *AOM '08: Proceedings of the 2008 AOSD Workshop on Aspect-Oriented Modeling*, pages 13–18, New York, NY, USA, 2008. ACM.
2. A. McNeile and E. Roubtsova. Executable Protocol Models as a Requirements Engineering Tool. In *ANSS-41 '08: Proceedings of the 41st Annual Simulation Symposium (anss-41 2008)*, pages 95–102, Washington, DC, USA, 2008. IEEE Computer Society.
3. A. McNeile and N. Simons. Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling* , 5(1):91–107, 2006.

4. A. McNeile, N. Simons. <http://www.metamaxim.com/>.
5. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
6. D. Dantas, D. Walker. Harmless Advice. *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. To appear, 2006.
7. E. Roubtsova and R. Kuiper. Process Semantics for UML Component Specifications to Assess Inheritance. ENTCS 73(3),Eds.P.Bottoni and M.Minas, 2003.
8. G. Engels, A. Kleppe, A. Rensink, M. Semenyak, C. Soltenborn and H. Wehrheim. From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations. In *ECMDA-FA*, pages 94–109, 2008.
9. H. Fecher, J. Schönborn, M. Kyas, W. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *ICFEM*, pages 52–65, 2005.
10. J. Ebert, G. Engels. Observable or invocable behaviour-You have to choose. *Technical report. Universitat Koblenz, Koblenz, Germany*, 1994.
11. J. Greenyer, J. Rieke, O. Travkin and E. Kindler. TGGs for Transforming UML to CSP: Contribution to the ACTIVE 2007 Graph Transformation Tools Contest. University of Paderborn, Technical Report tr-ri-08-287, 2008.
12. M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. www.w3.org/2002/ws/chor/edcopies/theory/note.pdf, 2006.
13. O. Sims. Presentation: MDA: The Real Value, Object Management Group website: www.omg.org/mda/presentations.htm . 2002.
14. OMG. Model Driven Architecture: How Systems Will Be Built. Object Management Group website: www.omg.org/mda/.
15. OMG. Unified Modeling Language, Superstructure, v2.2. *OMG Document formal/09-02-02 Minor revision to UML, v2.1.2. Supersedes formal 2007-11-02*, 2009.
16. R. Milner. *A Calculus of Communicating Systems*, volume 92. 1980.
17. R. Soley. Presentation: MDA: An Introduction. Object Management Group website: www.omg.org/mda/presentations.htm . 2002.
18. S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. 2002.
19. S. Shlaer and S. Mellor. *Object Life Cycles - Modeling the World in States*. Yourdon Press/Prentice Hall, 1992.
20. T. Santer and D. Seifert. Executing UML State Machines. *Technical Report 2006-04, Fakultt fr Elektrotechnik und Informatik, Technische Universitt Berlin*, 2006.
21. W. Grieskamp, F. Kicillof, N. Tillmann. Action Machines: A Framework for Encoding and Composing Partial Behaviours. *Microsoft Technical Report MSR-TR-2006-11*, 2006.

Towards a Model Execution Framework for Eclipse

Michael Soden and Hajo Eichler

Department of Computer Science, Humboldt University
 Unter den Linden 6, 10099 Berlin, Germany
 soden@ikv.de eichler@ikv.de

Abstract. The Eclipse Modeling Project (EMP) is one of the most striking foundation for model driven development. With its core frameworks for metamodeling, textual and graphical editors, validation & constraints, transformations, etc. it provides broad support for creation of model driven tooling such as for Domain Specific Languages (DSLs). However, there is currently a lack of support for making *models executable* by means of operational semantics. This paper outlines the M3Actions, a framework to develop execution semantics for MOF metamodels, which is on the verge of being adopted as basis of a new Eclipse project named *Model Execution Framework* (MXF). We discuss requirements upon model execution and sketch requirements of a common execution infrastructure.

1 Introduction

The areas of model driven engineering, language oriented programming and domain specific modeling have grown rapidly over the last years. For Java-based development, the Eclipse Modeling Project (EMP) [1] together with its sub-projects centered around the Eclipse Modeling Framework (EMF) [2] constitutes the *state-of-the-art* of model driven technology. While EMF models — as quasi standard implementation of the Meta Object Facility (MOF, [3][4]) — build the core of all modeling activities, supplementary components support the creation of tooling for models such as Xtext for textual editors, GMF for graphical editors, M2M for model transformations, OCL for constraints, and so on [5][6][1][7]. However, a missing piece under the umbrella of EMP is a framework to define execution semantics for models by means of specifying operational semantics [8] for new languages or building e.g. domain specific simulators [9].

In this paper we present the *M3Actions* framework that was designed to fill this gap [10]. M3Actions is a framework that supports operational semantics for EMF¹ models and that consists of:

- A graphical editor to define structure and behavior of (meta-)models enhanced by explicit instantiation

¹ The concepts are actually defined for MOF metamodels [4], but the implementation is based on EMF

- A generic interpreter and debugger to execute those definitions
- A trace recorder to record execution runs as traces

Lessons learned from research at the Humboldt University Berlin on language semantics (e.g. [11][12]) have influenced the concepts of the framework which addresses the need to have human readable, high-level, but precisely executable language definitions [13][14]. Beside the pure language definition and execution, one design rational was the creation of an open model execution environment to plug-in further components for dynamic and static model analysis by means of trace analysis, runtime verification and testing of models. Out of this work on executable models and discussions with people of the Eclipse community, the idea was born to establish a *Model Execution Framework* (MXF) for Eclipse [15]. MXF will realize a common model execution infrastructure as well as provide a framework for development, execution and debugging of models with operational semantics on the basis of M3Actions. At time of writing, the project has passed the creation review and is now in the approval process to adopt the M3Actions sources as initial contribution².

The remainder of this paper is organized as follows. Section 2 summarizes briefly the state of the art of model driven tooling at Eclipse, before section 3 introduces the extensions provided by M3Actions for model execution. Afterwards, we exemplify the concepts along a state-machine DSL in section 3.2. In section 4 and 5 we discuss related work and draw some conclusions on the foundation of MXF along a motivation of the project. Finally, section 6 provides an outlook on the future.

2 Modeling at Eclipse

In times of *The Unbearable Stupidity of Modeling*³, (meta-) models are used all-around software development with Eclipse. The trend of using models everywhere does not even pass by the core of Eclipse itself. There is a fundamental discussion to model all data structures used in all plug-ins of Eclipse with EMF for its next generation called *e4*⁴.

While the classic approach to mostly textual language development is rooted in the history of automata theory and parser technology, EMP is driven by higher-level concepts of object-oriented metamodeling [1]. Central point in building tools for domain specific languages is the definition of a metamodel with EMF (**ecore** model [2]), defining the data structure/abstract syntax. As notation of the metamodel, EMF offers a lot of representation forms like annotated source code, XML import as well as a tree-based and graphical editor⁵. Thereafter, a generator produces ad hoc deployable model repositories that store any instance

² for latest updates, please see the EMFT newsgroup (eclipse.technology.emft)

³ Ed Merks's Eclipse Summit 2008 presentation's title. He is the project lead of the EMF project

⁴ More information on e4 and modeling can be found at <http://wiki.eclipse.org/E4>

⁵ see EcoreTools component of EMFT at <http://www.eclipse.org/modeling/emft>

of the underlying metamodel as XMI (or custom XML). Together with supplementary components such as the EMF Edit Framework, OCL [16], etc. EMF enables fast and effortless out-of-the-box development of software components that store and manage domain specific data structures.

Additional frameworks established around EMF support the creation of tooling for a DSL. Since Eclipse offers a vast number of frameworks, we'll concentrate on two projects often used for the development of editors: Xtext [5] and GMF [6]. For the creation of a textual DSL, Xtext supports the generation of a full-featured text editor out of an annotated EBNF grammar. While previous versions rely on a derived metamodel for the AST, the latest version allows the generation of a parser that feeds a given EMF model repository. At time of writing, Xtext is on the transition of becoming the Textual Modeling Framework (TMF) of Eclipse. On the contrary GMF provides a generator and runtime infrastructure for the development of graphical editors for a given metamodel (also called domain model or semantic model). Graphical elements of the concrete syntax are defined by a *diagram definition model* and a *mapping model* that describes how elements of the concrete syntax are mapped to the semantic model. With these foundations the process to build tooling for a DSL is itself model-driven and round-trips are well supported by consistent re-generation of code.

Having built a DSL editor and potentially subsequent tooling, the next step would be to implement execution semantics into the editor to simulate the domain models. This is where the upcoming MXF project comes into action, targeting on provision of a common execution infrastructure and *to model* execution semantics instead of coding it [15].

3 The M3Actions Framework

Built on top of the Eclipse modeling projects, the M3Action framework provides extensions that allow the definition, execution and debugging of models with execution semantics (i.e. operational semantics). The core of the framework is the *MAction* language that consists of a number of elementary actions to define model manipulations. Borrowing its graphical syntax from UML2 Activities/Actions, flows are hooked into the metamodel as *MOperations*. Supported basic actions are:

MQueryAction executes an OCL query over the model and returns the result at an output pin.

MCreateAction Instantiates a specified meta-class and returns a new instance at an output pin.

MAssignAction modifies object properties, i.e. assigns, adds or removes values of single and multi-valued properties.

MInvocationAction invokes another operations with a given context object '*self*' or a *MActivity* (context-less flow of actions).

MIterateAction iterates a collection specified by an OCL query.

MAtomicGroup groups a set of actions to build more complex behavior which is semantically *atomic*

Behavior is always executed in the context of a `MThread`, and multi-threading is supported by the `MInvocationAction`.

A second aspect in the framework's architecture is the differentiation between the (fixed) abstract syntax of a language and its evolving runtime configurations. Runtime models are again defined by a metamodel (the *runtime metamodel*) and manipulated through actions. Note that although there is technically no difference between abstract syntax models and runtime models, changes shall only be applied to the runtime model for ease of having a clear separation between language concepts and execution concepts. Hence, all information required for model execution goes into the latter, for example value domains, program counters, data structures such as stacks-frames, and so on. The (meta-)models and their relationships are shown in figure 1.

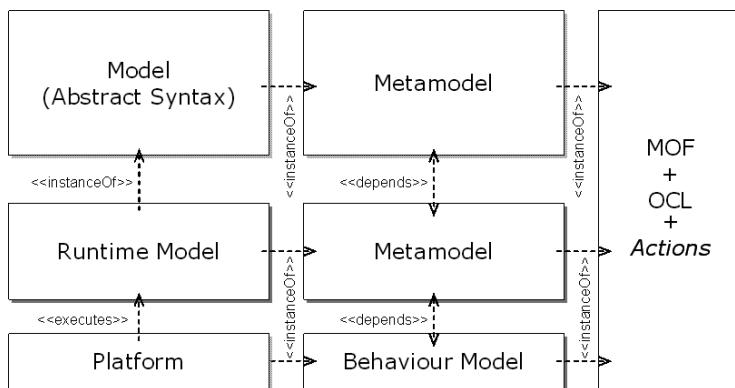


Fig. 1. Architecture: models and their meta-layer relationships

In addition to the common concepts of EMF, the M3Actions framework provides an explicit *instanceOf* relation for modeling multiple, logical meta-layers. Thereby, two classes can have an *instanceOf* relation, which links their physical instances logically together as 'object' and 'meta-object'. The concept is exemplified together with an example in section 3.2. For more details about the instantiation concept please refer to [14].

3.1 Controlling states

Designing the behavior of a language requires a precise specification of *observable states* if it comes to the analysis of execution runs. Execution of models can be recorded as a trace for each running thread. These execution traces are defined by a generic trace metamodel⁶ that is independent of a specific language metamodel. Each trace consists of a list of change objects that describe the delta of a model modification.

⁶ cf. to [10] for more details

Since actions change the runtime model, a new *micro-state* is reached whenever an action modifying the model is done. In most cases, states of the designed language do not match these micro-states of the action defining its behavior. For this purpose, the MActions provides two concepts to control states explicitly:

1. **Atomic Groups** - Atomic groups protect contained actions from being interrupted. Apart from threading, this language-construct combines all changes caused by contained actions and produces a single change-object in the trace.
2. **State Generating Transitions (SGTs)** - SGTs provide a means of specifying when a new state is entered. While normal transitions define the control- and data-flow, SGTs produce a new state whenever they are visited during execution.

As a direct consequence, the recorded traces reflect not only the states of the executed model (runtime model) but the language's *observable states*. These states are required for dynamic and static model analysis. We will explain more details along with the example DSL in section 3.2.

3.2 Example: DSL for StateMachines

This section illustrates the application of M3Actions along an example DSL: a finite state-machine language. The language was inspired by Martin Fowler's upcoming book on DSLs (currently published on his website [17]). The DSL defines a typical state/transition language with state-machines communicating via exchanged events. The approach how to define the execution semantics will be explained along a simple Ping-Pong example model shown in Figure 2.

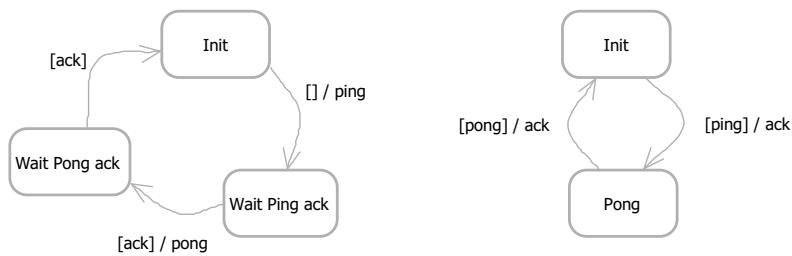


Fig. 2. The Ping-Pong state machine model: (right) sender, (left) responder

The overall system consists of two state-machines: *sender* and *responder*. For the DSL notation, we have build a GMF editor [6] that uses a UML-like notation for states and transitions with the following syntax for transition labels:

$$[<\text{guard}>] / <\text{action}> \quad (1)$$

where [`<guard>`] defines the event that triggers the transition and `<action>` specifies an event which is produced. By convention, initial states are defined using the name `Init`. Hence, the sender is the initiator of the whole communication since it has an outgoing transition with an empty guard '[]'. Firing this transition will send a `ping` event and set the machine into a wait for confirmation state ('Wait ping ack'). The responder will consume the `ping` event and acknowledge the receipt with an `ack` event to the sender. Thereafter, the sender will send a `pong` event and wait again for acknowledgment. Finally, if no events are lost, the responder also confirms this second event and brings the whole system back to its initial states.

The structure of the state machines is defined by the metamodel shown in figure 3 (blank classes). The definition of all concepts should be intuitive and straightforward, except that labels as defined in (1) are mapped as string values to corresponding attributes `<guard>` and `<action>` of meta-class `Transition`.

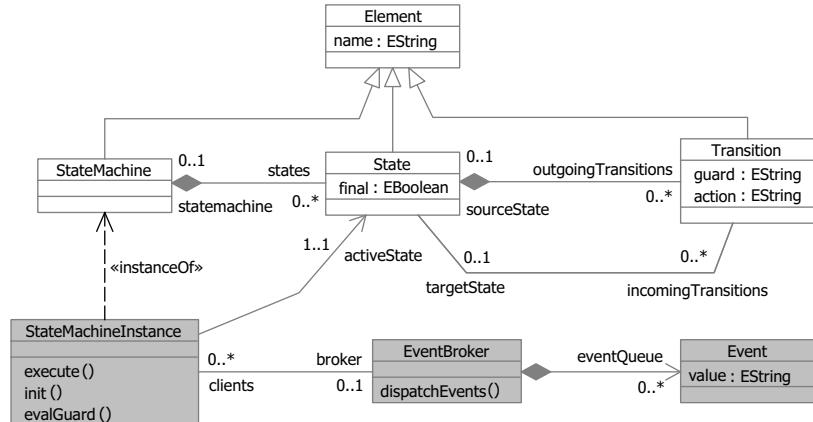


Fig. 3. Simple StateMachine Metamodel

Having defined the structural part of the DSL, we are able to specify the operational semantics of the state-machines precisely using *MActions*. First, we define the runtime model using three classes: `Event`, `EventBroker` and `StateMachineInstance` (marked gray in figure 3). Thereby, an instance of meta-class `StateMachineInstance` (abbreviated as `SMI` in the following) represents an *instance of* a `StateMachine`. In the *M3Actions* framework, this is expressed explicitly using the *instanceOf* relation. During runtime, this *instanceOf* relation is available and can be navigated via an implicit property `metaObject` (cp. section 3). As runtime data, the class carries only a reference to the current `activeState`. `SMIs` are connected to a 'platform infrastructure' represented by class `EventBroker`. Its task is to distribute `Events` to connected clients.

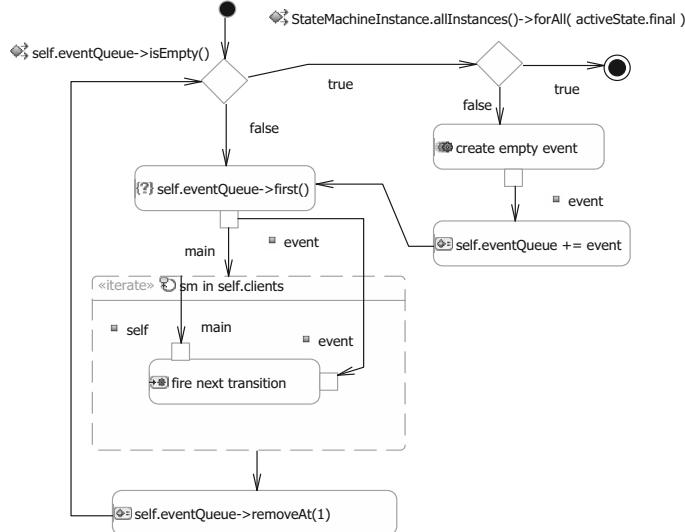


Fig. 4. EventBroker#dispatchEvents

The action semantics are hooked in via MOperations, depicted as operation signatures in figure 3. The main execution loop is contained in `dispatchEvents`, shown in figure 4. Assuming all SMIs are connected as `clients` to a broker instance, the main idea is to continuously broadcast each event to all client machines which in turn may emit events back that are scheduled by adding them at the end of the queue (FIFO). As seen in figure 4, first a decision node checks the `eventQueue` for emptiness. If the queue is empty, the termination condition checks whether all state-machines have reached a final state⁷. In case not all machines have reached a final state, a default event is created by a create action and afterwards it is enqueued in `eventQueue` with the multi-valued assign operator '`+=`'. The next action queries for the first element of this list and the flow continues with an iterate action over all `clients`⁸. The iteration over all connected clients constitutes the main part of the behavior and consists of a single invocation action named `fire next transition`. This action invokes `execute(event : Event)`, where the iteration variable `sm` is used as new `self` and the current `event` is passed in as parameter (`execute` will be discussed below). Once the iteration has finished, the processed event is removed from the queue using the multi-value `removeAt` operator⁹. Note that this global event dispatch behavior models the simplest form of a loss-free broadcast event channel.

⁷ If we assume that there is only a single event-broker, the check `self.clients->forAll(activeState.final)` would have been sufficient

⁸ Note that `main` indicates the control flow in case of multiple object flows

⁹ As in OCL, actions over collections use also indexes starting with 1

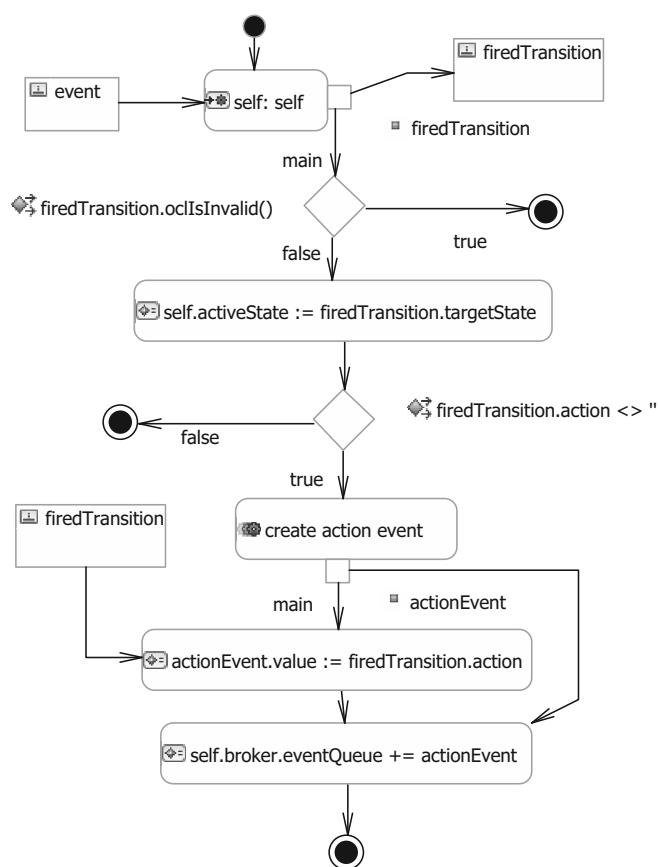


Fig. 5. StateMachineInstance#execute

The invoked behavior for each event is `execute` of class `SMI` as shown in figure 5. Briefly described, the behavior consists of three phases:

1. Evaluate the guards of the active state to find any transition that can fire for the current event. This is done by invoking `evalGuard` on '`self`' ('`self:self`' action has `evalGuard` as invocation target which is not shown in the diagram). The invoked behavior is shown in figure 6. Otherwise, if no transition is enabled, the behavior quits.
2. Once a transition is found (i.e. `firedTransition` holds a reference to that transition), `activeState` is updated to the target state of the transition.
3. In case an action clause is present at the transition, a new event is created with property `value` set to the `action` string of the fired transition. Finally, this new event is enqueued at the event broker.

The behavior of step (1) has been reduced to the minimum that is necessary to react on events. As can be seen in figure 6, only one decision node checks for enabled transitions by searching for a matching guard that is equal to the event's `value`. If such a transition is found, it is returned as `firedTransition`.

Note that one consequence of step (3) is that not every execution leads to the creation of events, but each registered state-machine *may* contribute a new event. In other words, the event queue may grow maximal by the number of registered clients in each iteration. Thus, dispatching only one event per cycle has the effect that the delivery of events might take longer the more traffic is caused by communication.

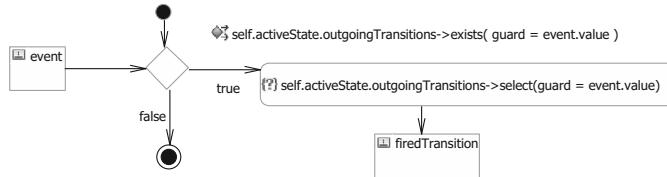


Fig. 6. StateMachineInstance#evalGuards

The three operations `dispatchEvents`, `execute` and `evalGuards` are all that is required as operational semantics and to execute e.g. the ping-pong example above. However, an initial system setup for instantiation and connection of the state-machines to the broker is necessary¹⁰. This startup behavior is shown in figure 7. After the event broker is instantiated, the iteration instantiates all given `StateMachines` by creating `StateMachineInstances` and connects them to the broker as clients. Thereby, the machine initialization takes place in the invocation action `instantiate machine` which simply sets the `activeState` reference to the state with name `Init` (cf. figure 8).

¹⁰ This might be solved differently by loading predefined runtime models directly, however, we define the complete setup for better comprehension and exercise here

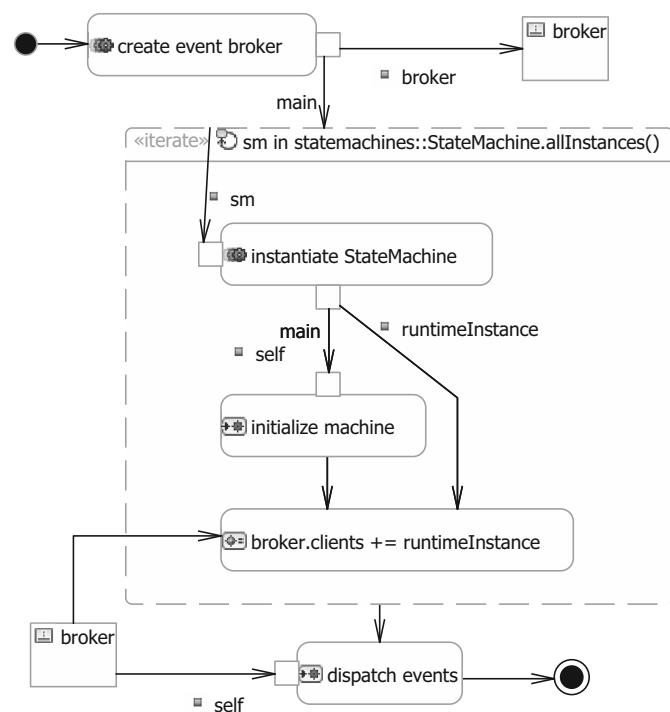


Fig. 7. Global startup behavior

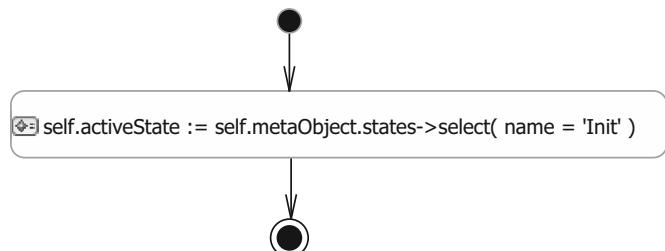


Fig. 8. StateMachineInstance#init

4 Related work

Today's notion of operational semantics goes back to the early work of Plotkin on *Structural Operational Semantics* [8]. While this formalization is bound to the concrete syntax, a further widespread approach to language semantics is the use of Abstract State Machines (ASMs [18][19]) which has been successfully applied to a number of languages including SDL[20], UML[21] or Java[22]. Beside these mathematical formalisms, newer generation languages and frameworks — evolving in different communities for language oriented programming or domain specific modeling — are based on the paradigm of *metamodeling*. On the one hand, these approaches are typically not based on a rigorous mathematical model, but provide on the other hand object oriented abstractions¹¹. M3Actions is a representative of such a *framework*, since it provides not only the MAction language to specify execution semantics¹², but also the embedding in an extensible framework with supplementary components (such as trace recording) for model analysis.

If we use the language dimensions concrete syntax (representation), abstract syntax (structure), static constraints and execution semantics (behavior, operational semantics) as done in [12], we can compare the various related approaches¹³ like XMF[25] or KerMeta[26] based on these categories. Although the listed approaches have a textual syntax language in contrast to the graphical syntax of MActions, all have in common their core *action language* to define the operational semantics inline within the metamodel. Beside basic operations to create, navigate, update or delete objects and properties which are mainly extensions to OCL, the approaches have different strength. XMF features additional library functions for I/O (via channels), UI dialogs and an elaborated support for syntax definitions through annotated grammars. In contrast, M3Actions has no such libraries per default, but provides the integration of extension actions that can be registered programmatically (this might be achieved using the Java interface of XMF). With respect to language features, MActions do not inherit the exception handling from UML which is e.g. present in KerMeta as `do..rescue` blocks. However, to the best of the authors knowledge, these languages do not support an explicit instantiation concept nor multi-threading¹⁴. Multiple threads of MActions are comparable to parallel ASMs [27]. Moreover, the explicit control over created observable states of the modeled DSL using `MAtomicGroup` and SGTs are a distinguished concept (cp. section 3.1).

Other frameworks and tools for language design are for example the ATLAS Model Management Architecture (AMMA, [28]), MetaEdit+ [29] or EProvide [30]. These frameworks and tools support different aspects of a DSL de-

¹¹ The development of the Abstract State Machine Language (AsmL [23]) is an example that provides both

¹² The authors are currently working on a formalization of MActions with ASMs

¹³ We've chosen these as typical representatives for related work even though there exist others, e.g. GME [24]

¹⁴ XMF claims to support multiple threads, but this is not exposed as concept in the language

sign, but fall back on 'external' approaches to define execution semantics. While AMMA uses the transformation language ATL [31] to define model changes, MetaEdit+ provides model animation through a scripting language plus API. More generally, the EProvide framework follows an open approach: it operates on a *big step semantic* for model changes and various languages for operational semantic can be plugged in.

5 Towards an Execution Framework

The various approaches for model execution were the main driver to initiate a common *Model Execution Framework* (MXF) for Eclipse [15]. Currently, the project is in an early stage bringing together experts from the field and their experience in order to build a core framework that supports execution of EMF models. The goal is to provide an integrated environment that supports interpreting and debugging of models with operational semantics and that serves as basis for further model analysis, assessment of models, testing, etc. Additionally, the project team aims for provision of an integration with GMF editors for building DSL simulators that are seamlessly integrated into a generated editor. For the individual parts of the framework, we have identified the following categories and requirements:

5.1 Runtime Models

It turns out that a primary artefact and precondition for model execution is the *runtime model* (or runtime information in general). Even though M3Actions have a specific view by regarding the runtime model as instance-of the abstract syntax model, related papers report also on the requirement of having these definitions beside the language metamodel itself (e.g. the *dynamic metamodel* of [32]).

5.2 Behavior Definitions

The MAction language and implementation is the proposed candidate to be used and/or extended for this purpose. Experiences with 'real languages' such as C# (cf. [33]) have shown, that such a graphical language scales well with respect to having clear and manageable diagrams. However, since we put the abstract syntax first, a textual syntax might be defined as an alternative representation of the same concepts. The project team considers to adapt the KerMeta textual syntax which is, at its core, close to MActions (cp. section 4).

5.3 Common Execution Infrastructure

The main aim is not to have a single all-purpose language that defines model execution semantics, however, but have a framework to integrate other languages by means of black-box operations, library implementations, and so on. It is the

authors' conviction that a closed language *will fail*, since many Eclipse users will continue to customize the execution through Java implementations in the same way as it is the case e.g. for model transformations or GMF editors. For example, a DSL simulator that involves mathematical calculations will definitively rely on a third party library (not even written in Java) on the implementation level. The common execution infrastructure will define common concepts on top of the Eclipse debugging framework and will enable applications to share runtime models, adapters for specific editors and debuggers, tracing capabilities, and more.

5.4 Concurrency

The MXF should provide a concept for parallelism similar to the threading concepts of M3Actions (not sketched in this paper, cf. to [10] for details). In summary, parallelism can be supported by either (1) modelling parallelism explicitly or (2) choose from an existing parallelism model of the meta-language. In the former case, one has to define thread queues, monitors, scheduling policies, and so on by oneself, whereas the latter provides reuse of common parallelism patterns. In the current implementation, we have investigated on three different concurrency models:

- true parallelism: an unlimited number of threads run at least conceptually *at the same time*.
- n-processor model: at most n parallel actions are executed at the same time.
- sequential execution: as special case of the n-processor model, all actions are enforced to be executed in a sequential order.

These concurrency models must be enforced by the execution environment, for example access to a global simulation timer will vary depending on the pattern chosen. In practice, this can lead to different outcomes with respect to a recorded execution trace where e.g. object changes of multiple threads have exactly the same time-stamp (simulation time instant).

5.5 Interpreter vs. Generator

While the existing interpreter and debugger works in a purely reflective manner (using EMF reflection capabilities), one goal is to develop language-specific simulator-/debugger-generators, optimizing runtime performance of the generic model interpreter. In the same conceptual line as EMF, the aim is to support re-generation of the execution logic whenever the language definition changes. As precondition, the generated code must fit exactly into the reflective execution infrastructure in the same way as e.g. the EMF reflection API behaves like the type-specific API.

6 Conclusion

This paper sketched existing Eclipse Modeling Projects that support the creation of DSL tools and motivated how valuable an extension for execution semantics of models is. To fill this gap, the M3Actions framework has been designed and provides operational semantics for EMF models. The framework is currently on the transition of becoming the base of the MXF project at Eclipse [15]. The strong interest on having a framework for executable model definitions is emphasized by feedback from the Eclipse community. As project leaders of the MXF project, the authors endeavor to bring together researchers and practitioners from the area of model execution and simulation in order to build a concise, but flexible execution framework for EMF.

References

1. Eclipse Project: (Eclipse Modeling Project (EMP), <http://www.eclipse.org/modeling/>) Last checked: January 15, 2009.
2. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework (The Eclipse Series). First edn. Addison-Wesley Professional (2003)
3. OMG: Meta Object Facility, Version 1.4. Object Management Group (2003) formal/2002-04-03.
4. OMG: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group (2003) ptc/03-10-04.
5. Efftinge, S., Völter, M.: oAW xText: A framework for textual DSLs. (2006)
6. Eclipse: (Graphical Modeling Framework, <http://www.eclipse.org/gmf/>) Last checked: January 15, 2009.
7. Eclipse: (Model-to-Model Transformation, <http://www.eclipse.org/m2m/>) Last checked: January 15, 2009.
8. Plotkin, G.: A structural approach to operational semantics. Technical report, University of Aarhus, Denmark (1981)
9. Valentin, E.C., Verbraeck, A.: Requirements for domain specific discrete event simulation environments. In: WSC '05: Proceedings of the 37th conference on Winter simulation, Winter Simulation Conference (2005) 654–663
10. Humboldt University Berlin: M3Actions - Operational Semantics for MOF Metamodels, <http://www.metamodels.de> (2008)
11. Prinz, A.: Formal Semantics for RSDL: Definition and Implementation. PhD thesis, Humboldt-Universität zu Berlin (2000)
12. Fischer, J., Holz, E., Prinz, A., Scheidgen, M.: Tool-based language development. In: Workshop on Integrated-reliability with Telecommunications and UML Languages. (2004)
13. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: ECMDA-FA. Volume 4530 of Lecture Notes in Computer Science., Haifa, Israel, Springer (2007) 157–171
14. Soden, M.: Operational semantics for MOF metamodels: Tutorial on M3Actions. <http://www.metamodels.de/docs.html> (2008)
15. Soden, M., Eichler, H.: Eclipse Proposal: Model Execution Framework, <http://www.eclipse.org/proposals/mxf/> (2009)
16. OMG: OCL 2.0 Specification. Object Management Group (2006) formal/2006-05-01.

17. Fowler, M.: DSL: An Introductory Example, (<http://martinfowler.com/dslwip/Intro.html>) Last checked: January 15, 2009.
18. Gurevich, Y.: Evolving algebras 1993: Lipari guide. (1995) 9–36
19. Gurevich, Y.: Abstract state machines: An overview of the project. Technical report, Microsoft Research (2003)
20. ITU-T: SDL formal definition: Dynamic semantics. In: Specification and Description Language (SDL). International Telecommunication Union (2000) Z.100 Annex F3.
21. Börger, E., Cavarras, A., Riccobene, E.: An ASM semantics for UML activity diagrams. In: AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology, London, UK, Springer-Verlag (2000) 293–308
22. Börger, E., Schulte, W.: A programmer friendly modular definition of the semantics of java. Technical report (1999)
23. Gurevich, Y., Rossman, B., Schulte, W.: Semantic essence of AsmL. Technical report, Microsoft Research (2004)
24. Agrawal, A., Karsai, G., Ledeczi, A.: An end-to-end domain-driven software development framework. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2003) 8–15
25. CETEVA: XMF. (<http://itcentre.tvu.ac.uk/clark/xmf.html>)
26. Team, T.: (Triskell Meta-Modelling Kernel. IRISA, INRIA. [www.kermeta.org.](http://www.kermeta.org/))
27. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms: Correction and extension. ACM Trans. Comput. Logic **9**(3) (2008) 1–32
28. Davide Di Ruscio, Frederic Jouault, I.K.J.B.A.P.: Extending amma for supporting dynamic semantics specifications of dsls. Technical report, Universite Studi dell'Aquila (2006)
29. MetaCase: (MetaEdit+. <http://www.metacase.com>.)
30. Sadilek, D.A., Wachsmuth, G.: Prototyping visual interpreters and debuggers for domain-specific modelling languages. In: ECMDA-FA. Lecture Notes in Computer Science, Springer (2008)
31. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: Atl: a qvt-like transformation language. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM (2006) 719–720
32. Combemale, B., Crégut, X., Giacometti, J.P., Michel, P., Pantel, M.: Introducing simulation and model animation in the MDE Topcased toolkit. In: European Congress on Embedded Real-Time Software (ERTS 2008), Toulouse, 29/01/2008–01/02/2008, Société des Ingénieurs de l'Automobile (2008)
33. Soden, M., Eichler, H.: An approach to use executable models for testing. In: Enterprise Modelling and Information Systems Architectures - Concepts and Applications , Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures. Volume P-119 of LNI., GI (2007)

Towards Model Structuring Based on Flow Diagram Decomposition

Arend Rensink, Maria Zimakova

Department of Computer Science, University of Twente,
P.O. Box 217, 7500 AE, The Netherlands
 {a.rensink, m.v.zimakova}@utwente.nl

Abstract. The key challenge of model transformations in model-driven development is in transforming higher-level abstract models into more concrete ones that can be used to generate implementation level models, including executable business process representations and program code. Many of the modelling languages (like UML Activity Diagrams or BPMN) use unstructured flow graphs to describe the operation sequence of a business process. If a structured language is chosen as the executable representation, it is difficult to compile the unstructured flows into structured statements. Even if a target language structure contains *goto*-like statements it is often simpler and more efficient to deal with programs that have structured control flow to make the executable representation more understandable.

In this paper, we take a first step towards an implementation of existing decomposition methods using graph transformations, and we evaluate their effectiveness with a view to readability and essential complexity measures.

Keywords. Model transformations, graph transformations, model structuring, flow diagram decomposition, data flow graph, complexity measure.

1 Introduction

Over the last few years, a new option has evolved to define solutions in software industry: Model-Driven Development (MDD). The key challenge of model transformations in MDD is in transforming higher-level abstract models into more concrete ones that can be used to generate implementation level models, including executable business process representations and program code. With this trend, the decomposition of the models into structured elements is of increasing importance.

In the large, a number of motivations can be given to justify the implementation of this work:

- Imagine a dynamic behaviour of business process is described as an unstructured flow graph (which can represent, by-turn, a UML activity or BPMN diagrams). If a structured language is chosen as the target executable representation, it is difficult to transform the unstructured flows into structured statements. This problem is analyzed, for instance, in [6] and attempts to

compile UMLA to BPEL programs; the last issue is discussed, for instance, in [16]. The main task of our graph transformations is to translate the unstructured *goto*-like statements into well-structured statements in the target language.

- The second very important reason for the presented work is to improve software reliability and readability – making programs less error prone and easier to understand. Because understanding of behavior is an essential prerequisite to effective program development and modification, programmers are forced to devote substantial time to this task [3].

There exists today a number of variants on the idea of well-structured models. A lot of restructuring methods were done in the context of flow diagram decomposition. It is commonly agreed that a natural interpretation of flow diagrams is in terms of *graphs* – essentially, just nodes with connecting edges. Consequently, a most natural implementation of flow diagram decomposition methods is by *graph transformations*.

The aim of this work is to bridge the gap between formalism of the existing flow diagram decomposition methods and practical implementation in terms of graph transformations to use it for modern programming environments including executable business process languages.

The remainder of this paper is structured as follows: after providing the basic definitions to set the stage in Section 2, we discuss the flow graph decompositions and complexity measure problem in Section 3. We consider these to be the heart of our contribution. In Section 4 we implement those methods with graph transformations, employing the graph-transformation tool Groove [14] for rule execution. Finally, in the conclusion (Section 5) we come back to the above considerations, evaluate our results and discuss plans for future work.

2 Basic Notions

Graphs and flow graphs. One of the core concepts of this paper is that of *graphs*. We start by repeating the usual definition of a graph.

Definition 1. A *labeled directed graph* is a tuple $G = (N, E, \lambda)$ where

- N is a finite nonempty set called a set of nodes;
- $E \subseteq N \times \Lambda \times N$ is a set of edges where Λ is a finite set of node and edge labels;
- λ is a labeling function $\lambda: N \cup E \rightarrow \Lambda$.

Given $e = (v, a, w) \in E$, we denote $src(e) = v$, $tgt(e) = w$ and $a = \lambda(e)$ for its source, target and label, respectively. A *path* in a graph G is an alternating sequence of nodes and edges beginning and ending with nodes such that for each $i \geq 1$ we have $v_i \in N$, $e_i \in E$, $src(e_i) = v_i$ and $tgt(e_i) = v_{i+1}$.

Let G be a labeled directed graph as above with a labeling function $\lambda: N \cup E \rightarrow \Lambda$, then a path $p = \{v_1, e_1, v_2, e_2, \dots, v_{k-1}, e_{k-1}, v_k\}$ in G can be represented by the *word* from the *alphabet* Λ as following:

$$\lambda(p) = \lambda(v_1)\lambda(e_1)\lambda(v_2) \dots \lambda(v_{k-1})\lambda(e_{k-1})\lambda(v_k).$$

We call this the *word representation* of p .

Definition 2. A *flow graph* Φ is a triple (G, s, t) , where

- $G = (N, E, \lambda)$ is a connected labeled directed graph;
- Node $s \in N$ is the unique start node such that there are no incoming edges to s in G .
- Node $t \in N$ is the unique terminal node such that there are no outgoing edges to t in G .

Figure 1 shows the simple example of a flow graph graphical representation, which will be used throughout this paper, because it contains most of the features needed to explain the transformation algorithms.

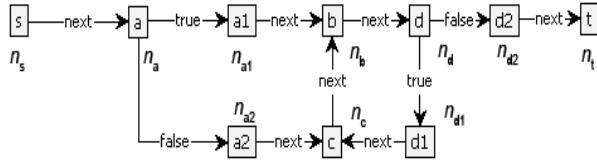


Fig. 1. Flow graph example

There are two most common types of nodes in a flow graph:

- The *functional type (function)* which represent some operations (semantically described by label $\lambda(n)$) to be carried out on an object $v \in N$.
- The *predicative type (predicate)* which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of $v \in N$.

In this paper we distinguish functional and predicative node types by count of their leaving edges as follows: the functional box can has just only one leaving edge (with *next* label for our example in Figure 1) and the predicative box can has just only two leaving edges (with *true* and *false* labels for our example in Figure 1).

The different node types that are supported by flow graphs, together with their relationships, are shown in Figure 3 (a), where we appeal to the reader's intuition about the meanings of this graph.

Let $\Phi = (G, s, t)$ be a flow graph and p be some path in G from the start node s to the terminal node t . Then we will say that p is a *full path* in the flow graph Φ .

Definition 3. Let p be a full path in a flow graph $\Phi = (G, s, t)$. Then an *execution sequence* $\text{Seq}(p)$ is the word representation of p .

For instance, sequence $(s \text{ next } a \text{ true } a_1 \text{ next } b \text{ next } d \text{ false } d_2 \text{ next } t)$ is an execution sequence for our example in Figure 1.

Now, let Path be a set (maybe infinite) of all possible full paths in the flow graph $\Phi = (G, s, t)$ in the light of the discussion above. The word representation of Path thus regarded as a *language* $\text{Lang}(\Phi) = \lambda(\text{Path})$ defined over the alphabet Λ .

Definition 4. Two flow graphs Φ_1 and Φ_2 are *equivalent* (denote it as $\Phi_1 \sim \Phi_2$) if they define the same languages: $\text{Lang}(\Phi_1) = \text{Lang}(\Phi_2)$.

Algebra of flow diagrams. A flow diagram is a graphical representation of the flow graph which is suitable for representing programs, Turing machines, etc. Diagrams are usually composed of boxes connected by directed lines.

Following [2], we can distinguish three elementary types of flow diagrams Π , Δ and Ω which denote, respectively, the diagrams of Figure 2 (a)-(c) and the constructions ‘sequence’, ‘if-then-else’ and ‘while’ in programming languages. Let us call these four elementary types $\Gamma = \{\Pi, \Delta, \Omega\}$ base subdiagrams.

For our subsequent definitions we also use the notions of a signature and algebra, as defined in [5]. The ingredients of these definitions that are important here are:

- A collection of data *sorts* *Sort*.
- A collection of carrier sets *Data*, partitioned into subsets for each of the sorts in *Sort*.
- A mapping *par*: *Oper* \rightarrow *Sort*⁺ that associates to every operation *op* \in *Oper* a non-empty string of sorts.

Note that an operation *op* with no parameters represents a constant value.

Let us assume a universe Θ of arbitrary flow graphs, a set $\theta_{func} \subset \Lambda$ of all functional node labels and a set $\theta_{pred} \subset \Lambda$ of all predicative node labels.

Definition 5. Let $\Phi = (G, s, t)$ be an arbitrary flow graph where $G = (N, E, \lambda)$ and $N' = N \setminus \{s, t\}$. A *flow graph substitution* is a mapping $\text{Sub}: N' \rightarrow \Theta$ that maps each node $v \in N'$ to a flow graph $\Phi_v = (G_v, s_v, t_v)$ where $G_v = (N_v, E_v, \lambda_v)$, and obeys the following rules:

- $\Phi[\Phi_v / v] = (G_{\text{Sub}}, s_{\text{Sub}}, t_{\text{Sub}})$ is a flow graph, $G_{\text{Sub}} = (N_{\text{Sub}}, E_{\text{Sub}}, \lambda_{\text{Sub}})$, $s_{\text{Sub}} = s$ and $t_{\text{Sub}} = t$;
- $N_{\text{Sub}} = (N \setminus v) \cup (N_v \setminus \{s_v, t_v\})$;
- $E_{\text{Sub}} = (E \setminus E^{\text{Del}}) \cup (E_v \setminus (E_v^{\text{Del}}) \cup (E_s^{\text{Ins}} \cup E_t^{\text{Ins}}))$ where
 - $E^{\text{Del}} = \{e \in E: \text{src}(e) = v \text{ or } \text{tgt}(e) = v\}$,
 - $E_v^{\text{Del}} = \{e_v \in E_v: \text{src}(e_v) = s_v \text{ or } \text{tgt}(e_v) = t_v\}$,
 - $E_s^{\text{Ins}} = \{e_{\text{Sub}} \in E_{\text{Sub}} \mid \exists e_v \in E_v: \text{src}(e_v) = s_v, \text{tgt}(e_v) = \text{tgt}(e_{\text{Sub}}), \lambda(e_v) = \lambda(e_{\text{Sub}}); \exists e \in E: \text{src}(e) = \text{src}(e_{\text{Sub}}), \text{tgt}(e) = v\}$,
 - $E_t^{\text{Ins}} = \{e_{\text{Sub}} \in E_{\text{Sub}} \mid \exists e_v \in E_v: \text{src}(e_v) = \text{src}(e_{\text{Sub}}), \text{tgt}(e_v) = t_v, \lambda(e_v) = \lambda(e_{\text{Sub}}); \exists e \in E: \text{src}(e) = v, \text{tgt}(e) = \text{tgt}(e_{\text{Sub}})\}$.

A substitution Sub can be extended to the whole flow graph as

$$\Phi[\text{Sub}] = \Phi[\Phi_{v_1} / v_1][\Phi_{v_2} / v_2] \dots [\Phi_{v_n} / v_n].$$

Let us define the signature $\text{Sig} = (\text{Sort}, \text{Oper}, \text{par})$ for the flow graphs. We have sorts *fg*, *pred* and *func*, representing the arbitrary flow graphs, predicative nodes and functional nodes, respectively. We also define a constant *empty* for the empty flow graph and operation symbols for the elementary flow graphs (for each functional node) and the base subdiagrams $\Gamma = \{\Pi, \Delta, \Omega\}$:

Sig =
Sort: *fg, pred, func*;
Oper: *empty, elem, Π, Δ, Ω*;

par: $\text{empty} : \rightarrow \text{fg}$,
elem: $\text{func} \rightarrow \text{fg}$,
 $\Pi : \text{fg fg} \rightarrow \text{fg}$,
 $\Delta : \text{pred fg fg} \rightarrow \text{fg}$,
 $\Omega : \text{pred fg} \rightarrow \text{fg}$.

Then the implementation of the signature `Sig` for flow graphs is the following algebra `FlowGraph`:

$$\begin{aligned}
D_{\text{fg}} &= \Theta, \\
D_{\text{func}} &= \theta_{\text{func}}, \\
D_{\text{pred}} &= \theta_{\text{pred}}, \\
f_{\text{empty}} &= \varepsilon \in \Theta, \\
f_{\text{elem}} &: D_{\text{func}} \rightarrow D_{\text{fg}}, \\
&\quad a \mapsto \{(N, E, \lambda) \mid N = \{s, v, t\}, E = \{(s, l, v), (v, l, t)\}, \\
&\quad \quad \quad \lambda(v) = a\} \\
f_{\Pi} &: D_{\text{fg}} \times D_{\text{fg}} \rightarrow D_{\text{fg}}, \\
&\quad (\Phi_a, \Phi_b) \mapsto \Pi[\Phi_a / v_a][\Phi_b / v_b] \\
f_{\Delta} &: D_{\text{pred}} \times D_{\text{fg}} \times D_{\text{fg}} \rightarrow D_{\text{fg}}, \\
&\quad (\alpha, \Phi_a, \Phi_b) \mapsto \Delta[\Phi_a / v_a][\Phi_b / v_b] \\
f_{\Omega} &: D_{\text{pred}} \times D_{\text{fg}} \rightarrow D_{\text{fg}}, \\
&\quad (\alpha, \Phi_a) \mapsto \Omega[\Phi_a / v_a].
\end{aligned}$$

Definition 6. A flow diagram $\Phi = (G, s, t)$ where $G = (N, E, \lambda)$ is *strongly decomposable* (or *well-formed* in terms of [6] and [13]) if there exists an expression exp in the `Sig`-algebra `FlowGraph` such that $\text{FlowGraph}[\text{exp}] \cong \Phi$.

Together with a strong decomposition, [2] considered another decomposition which is obtained by operating on an *equivalent* strongly decomposable flow graph. Formally, a flow graph Φ is *weakly decomposable* if $\Phi \sim \Phi'$ for some strongly decomposable flow graph Φ' .

Algebra of syntax trees. The other data structure for representing programming language constructs by compilers, converters and transformation tools is a tree structure known as an *abstract syntax tree* [11].

In terms of graph theory, an abstract syntax tree is a tree, that is to say, an acyclic graph with a single root node, connecting nodes and leaf nodes. Then, similarly to the graph definition above, we can define a syntax tree as follows.

Definition 7. An *abstract syntax tree*, or just *syntax tree*, is a tuple $T = (G_T, \text{root})$ where

- $G_T = (N_T, E_T, \lambda_T)$ is an acyclic connected labeled directed graph;
- $\text{root} \in N_T$ is a single root node;
- $N_T = N_n \cup N_l$ such as $N_n \cap N_l = \emptyset$ where N_n is a set of *internal nodes* and N_l is a set of *leaf nodes*.

Each node of the syntax tree in our case should denote a construction occurring in the flow diagram. For instance, the base subdiagrams in Figure 2 (a)-(c) may be denoted by constructions `Seq`, `IfThenElse` and `While` in Figure 2 (d)-(f),

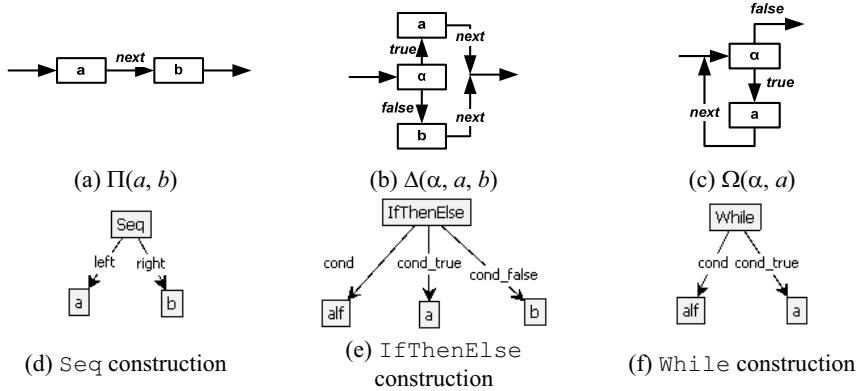


Fig. 2. Diagrams of $\Pi(a, b)$, $\Delta(a, a, b)$, $\Omega(a, a)$ and respective syntax tree construction

respectively. The different node types that are supported by syntax trees, together with their relationships, are shown in Figure 3 (b).

Similarly to the algebra `FlowGraph` above, we can implement a signature `Sig` with a different algebra `SyntaxTree` on a set ϑ of syntax tree constructions $\{\text{Seq}, \text{IfThenElse}, \text{While}\}$.

Let us consider now a representation of a flow graph Φ as a syntax tree T , called a *syntax tree decomposition*.

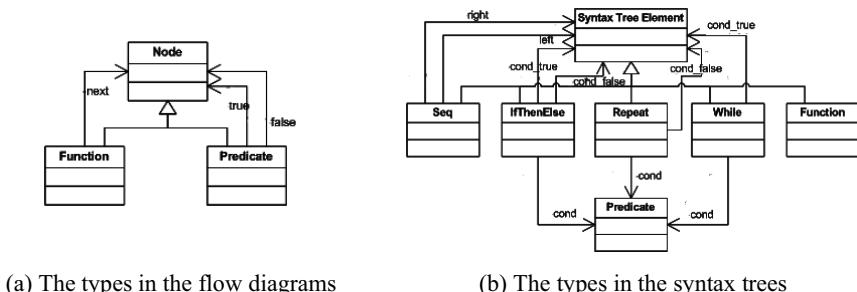
Definition 8. A *syntax tree decomposition* of a weakly decomposable flow graph $\Phi = (G, s, t)$ is a following morphism:

$$STD: \Phi \mapsto \{\text{SyntaxTree}[\text{exp}] \mid \text{FlowGraph}[\text{exp}] \cong \Phi' \sim \Phi\}$$

where $\Phi' = (G', s, t)$ is an strongly decomposable flow graph equivalent to Φ .

3 Flow Diagram Decomposition

In this section we consider the structuring problems imposed by our common example. The first problem is a very complicated graph structure of the flow diagram.



(a) The types in the flow diagrams

(b) The types in the syntax trees

Fig. 3. The types in the flow diagrams and syntax trees

One of the decomposition approaches to solve that kind of problem was provided in [2]. We discuss details of this approach in Section 3.1. The main flow diagram also has one meaningful (more than one node) *strongly connected component* (SCC); therefore we can improve the decomposition quality applying the method of [13]. The application of this algorithm as a part of the general approach is discussed in Section 3.2. The complexity measure to evaluate the advantage of different methods is considered in Section 3.3 and some concrete implementation results are presented in Section 4.

A concise review of many of other results developed in this field has been prepared in [7]. We also come back to that discussion in the closing remarks about future work in Section 5.

3.1 Base subdiagram decomposition

The set of definitions introduced in the previous section is within the scope of the existing graph theory. In this section, we introduce a way to enrich the usual definitions, and so formalize the concepts of flow graph decomposition.

The preliminaries of Böhm-Jacopini method [2] were presented in Section 2. In addition to three base subdiagrams Π , Ω and Δ , they introduced three new functions denoted by T , F , K , and a new predicate ω which define a behavior of auxiliary boolean variables set.

The effect of the first two functions T and F is to create a new boolean variable with value *true* or *false*, respectively, and the function K deletes the last boolean variable. The predicate ω is verified or not according to whether the last boolean variable value is *true* or *false*; the value of the predicate ω is *true* iff the last boolean variable value is *true*.

Recall that if Path is a set of all possible full paths in the flow graph Φ , then the word representation of Path can be regarded as a language $\text{Lang}(\Phi)$ defined (in the extended case) over the alphabet $\Lambda \cup \{T, F, K, \omega\}$. Let the node types and their relationships be as it shown in Figure 3 (a).

Then we can define a ‘*satisfiability*’ function $\text{Sat}: \text{Lang}(\Phi) \rightarrow \text{Lang}^*(\Phi)$, where $\text{Lang}^*(\Phi) = \text{Lang}(\Phi) \cup \{\epsilon\}$, as following: for all words $w = (x_1 x_2 \dots x_i \dots x_j \dots x_n) \in \text{Lang}(\Phi)$ where $x_k \in \Lambda \cup \{T, F, K, \omega\}$, $k \in [1, n]$

$$\text{Sat}(w) = \begin{cases} \epsilon & \text{if } \exists i, j \in [2, n-2], i < j : \\ & x_i \in \{T, F\}; x_j = \omega; \\ & x_{j+1} \in \{\text{true}, \text{false}\} \setminus \{\tilde{x}_i\} \text{ and} \\ & \forall k \in [i+1, j-1]: x_k \notin \{T, F, K, \omega\}; \\ w & \text{otherwise} \end{cases} \quad \text{where } \tilde{x} = \begin{cases} \text{true} & \text{if } x = T; \\ \text{false} & \text{if } x = F; \\ x & \text{otherwise} \end{cases} .$$

Therefore the language $\text{Sat}(\text{Lang}(\Phi))$ denotes a set of all full path word representations in the flow graph Φ that satisfy our definitions of new functions T , F , K and predicate ω .

Let us denote a function $\text{Restrict}: \text{Lang}^*(\Phi) \rightarrow \text{Lang}^*(\Phi) \setminus \{T, F, K, \omega\}$ as following: for all words $w = (x_1 x_2 \dots x_{i-1} x_i x_{i+1} x_{i+2} \dots x_n) \in \text{Lang}^*(\Phi)$ where $x_i \in \Lambda$,

$j = 1, 2, \dots, i-1, i+1, \dots, n$ and $x_i \in \{T, F, K, \omega\}$

$$\text{Restrict}(w) = (x_1 x_2 \dots x_{i-1} x_{i+2} \dots x_n).$$

Then a language $\overline{\text{Lang}}(\Phi) = \text{Restrict}(\text{Sat}(\text{Lang}(\Phi)))$ is a *restricted language* of the flow graph Φ over the alphabet Λ . Then we can extend the definition of flow graph equivalence.

Definition 9. Two flow graphs Φ_1 and Φ_2 extended by functions T, F, K and predicate ω are *equivalent* if they define the same restricted languages, that is $\overline{\text{Lang}}(\Phi_1) = \overline{\text{Lang}}(\Phi_2)$.

In the light of this discussion above the definition of weak decomposition can be extended as a decomposition which is obtained by operating on an equivalent strongly decomposable *extended* flow graph.

Theorem 1. For any flow graph Φ_1 there exists (at least) one equivalent strongly decomposable flow graph Φ_2 extended by the functions K, T, F and predicate ω ; in other words, any flow graph is weakly decomposable.

The proof of the theorem and the decomposition algorithm is based on the flow diagram classification represented in Figure 4 (a)-(c). The equivalent strongly decomposed flow diagram of type I is shown in Figure 5 (for more details see [14]).

3.2 SCC decomposition

Peterson *et al.* present the algorithm enabled to improve characteristics of Böhm-Jacopini method in case if flow graph consists of strongly connected components with multiple entry points [13].

Theorem 2. Every flow diagram can be transformed into an equivalent strongly decomposable (well-formed) flow diagram by node duplication (proof see [13]).

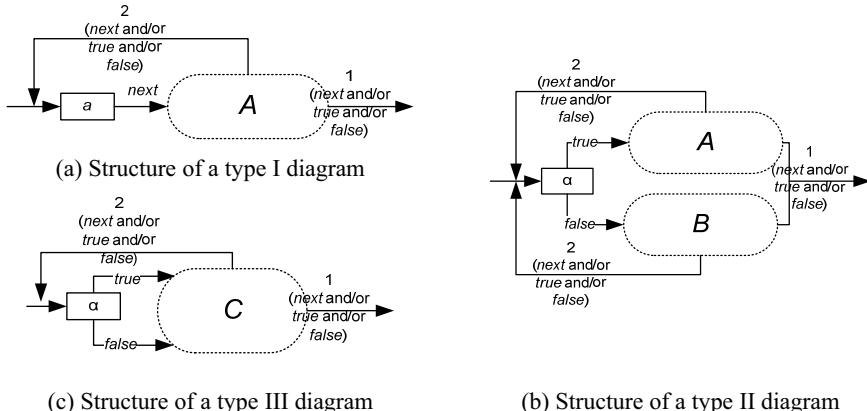


Fig. 4. Three types of flow diagrams

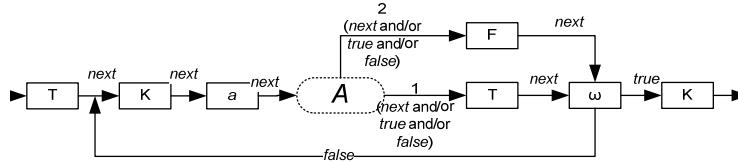


Fig. 5. Transformation of a type I diagrams

In the proof of this theorem the authors presented the algorithm that examines strongly connected components for multiple entry points and removes extra entry points by node duplication.

Let us come back to our main example in Figure 1 where nodes b , c , d and d_1 form a strongly connected component, and b and c are multiple entry nodes. If b is chosen as the entry node and c is duplicated, the well-structured flow diagram with the extended flow graph shown in Figure 6 (c) results. This turns out to be the better choice because this flow graph is intuitively ‘better’ than the flow graph in Figure 6 (a).

But if c is chosen as the entry node and b is duplicated, some more duplicating steps are necessary, and after four steps we can obtain the same flow graph as in Böhm-Jacopini method shown in Figure 6 (a), as well as three different flow graphs not shown here.

The fact that there are many variants of equivalent flow graphs, and some of them are ‘better’ than another, brings us to the issue of *complexity measuring* presented in the next section.

3.3 Complexity measuring

Maintenance typically requires more resources than new software development. For years researchers have tried to understand how programmers comprehend programs. The literature provides two approaches to comprehension: cognitive models that emphasize cognition by what the program does (a functional approach) and a control-flow approach which emphasizes how the program works. A modern state of the art of this direction is reflected in the review [3].

A well-known and often used complexity measure was proposed by McCabe in [10].

Definition 10. The *cyclomatic number* $v(\Phi)$ of flow graph Φ with n nodes, e edges, and p connected components is

$$v(\Phi) = e - n + 2p.$$

In addition, McCabe proposed a method of measuring the “structuredness” of a program as follows.

Let a *decomposition degree* $m(\Phi)$ of a flow graph Φ be a number of substitutions Φ_{v_i} , $i = 1, \dots, n$, such that $\Phi_{v_i} \in \Gamma \setminus \{\Pi\}$. Then

Definition 11. The following definition of *essential complexity* $v_e(\Phi)$ is used to reflect the lack of structure:

$$v_e(\Phi) = v(\Phi) - m(\Phi).$$

In the large, we propose to measure a full complexity of the flow diagram as follows:

Definition 12. Let $v(\Phi)$ be the cyclomatic number, $v_e(\Phi)$ - the essential complexity number and $v_d(\Phi)$ - the number of duplicated nodes in a flow graph Φ . Then the following defines the *full complexity* $V(\Phi)$:

$$V(\Phi) = [v(\Phi) + v_d(\Phi)] \times v_e(\Phi).$$

This formula stresses that the full complexity of a flow diagram is equal to the summation of its cyclomatic number and number of duplicates. The multiplication dictates that the full complexity and essential complexity of a flow diagram must be in the same order of magnitude.

Let us illustrate all of that complexity measuring by our main example shown in Figure 1. The initial flow diagram contains two predicates, therefore $v = 3$, $v_e = 3$, $v_d = 0$ and $V = (3 + 0) \times 3 = 9$. If we apply the straight Böhm-Jacopini method the final flow diagram shown in Figure 6 (a) has $v = 6$, $v_e = 1$, $v_d = 4$ and $V = (6 + 5) \times 1 = 11$. The ‘best choice’ of SCC method represented in Figure 6 (c) has $v = 4$, $v_e = 1$, $v_d = 1$ and $V = (4 + 1) \times 1 = 5$. Other four flow graphs obtained by SCC method have $V = 6$, $V = 11$, $V = 12$ and $V = 12$, respectively.

Hereby, the introduced full complexity measure V reflects an intuitive notion of readability and enables us to compare the final syntax trees and minimize their complexity.

4 Groove Implementation

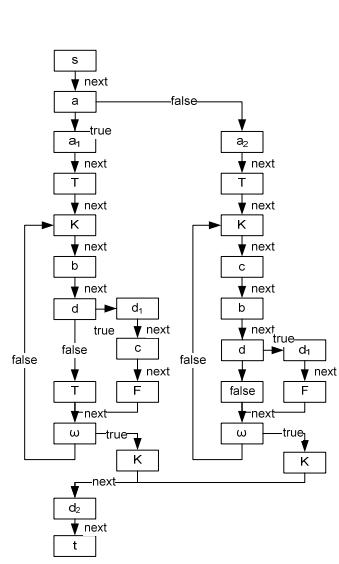
We implemented techniques described in Section 3 within the Groove (see [5], [9], [14]) framework, a standard tool for graph transformations. This allowed a more thorough exploration of more examples and for a qualified judgment on practical scalability.

The flow diagram decomposition rules construct a syntax tree by contracting and transforming a flow diagram. In this transformation process, syntax tree elements are introduced to the flow diagram and flow diagram elements are contracted (iteratively) to one node. Our flow diagram decomposition approach consists of following issues:

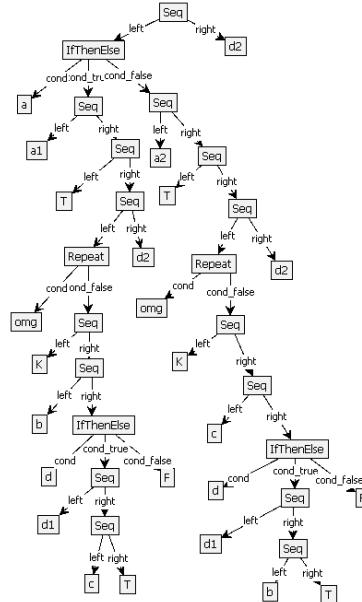
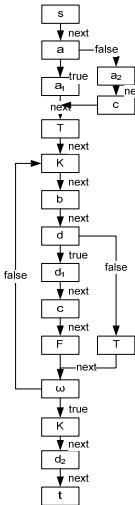
- *Flow diagram and syntax trees.* On the first step of our transformations we copy the initial flow diagram Φ to create the same structure for the syntax tree T .
- *Contraction rules.* For each type of elementary flow diagrams Π , Ω and Δ , we design one flow diagram *contraction rule* that introduce the necessary syntax tree elements and contracts elementary flow diagram to one node.
- *Decomposition rules.* The flow diagram *decomposition process* operates top-down, starting from the root-node of the flow diagram under construction and choosing an appropriate type of flow diagram as was discussed in Section 3.1.
- *SCC rules.* To improve readability of the flow diagrams, we also use *strongly connected component (SCC) decomposition rules* as it was discussed in Section

3.2.

- *Bottom-up and top-down decomposition.* In general, the flow diagram contraction and decomposition process operates in both directions: while an



(a) Böhm-Jacopini decomposition

(b) The syntax tree decomposition of graph (a) with $V = 11$ 

(c) Decomposition using SCC method

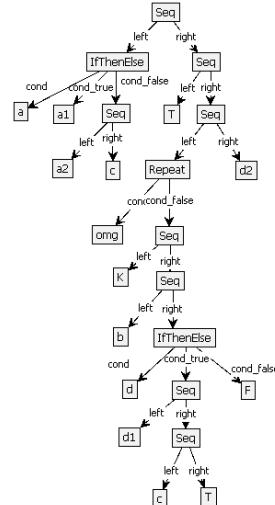
(d) The syntax tree decomposition of graph (b) with $V = 5$

Fig. 6. Two strongly decomposable (well-formed) extended flow graphs equivalent to the flow graph in Fig. 1 and respective final syntax trees.

extraction of elementary flow diagram is possible, we are applying one of contraction rules and have a *bottom-up* process; otherwise we are applying one of decomposition rules and have a *top-down* decomposition.

- *Syntax trees*. On the last step of our transformation we delete the contracted flow diagram elements and get a final syntax tree.

Unfortunately, we cannot explain the precise workings of the Groove implementation in the available space; however, the rules and some example cases are available at [15] for the reader to try out.

The example of the final syntax tree for the straight Böhm-Jacopini method applied to the initial flow diagram in Figure 1 is shown in Figure 6 (b) and has $v = 6$, $v_e = 1$, $v_d = 4$ and $V = (6 + 5) \times 1 = 11$. The best of five final syntax trees corresponding to that initial diagram obtained by the nondeterministic SCC method (see Section 3.2) is shown in Figure 6 (d) and has $v = 4$, $v_e = 1$, $v_d = 1$ and $V = (4 + 1) \times 1 = 5$. The text code representation corresponding to the final syntax trees in Figure 6 (b) and Figure 6 (d) are presented in Figure 7 (a) and Figure 7 (b), respectively.

Some example results for the complexity measuring implementation are given in Table 1. From the table, we can observe that (as expected) the SCC method always yields results at least as good as, and in all larger cases better than, the Böhm-Jacopini method. The detailed description of examples is available at [15].

Two flow graphs with 50 and 100 random nodes and edges are interesting as performance and scaling test cases. The results comprise about 1500 and 2500

```

begin
  if a then begin
    a1;
    var_bool := true;
    repeat
      b;
      if d then begin
        d1; c;
        var_bool := false;
      end else
        var_bool := true;
    until var_bool;
  end else begin
    a2; var_bool := true;
    repeat
      c; b;
      if d then begin
        d1;
        var_bool := false;
      end else
        var_bool := true;
    until var_bool;
  end;
  d2;
end.

begin
  if a then
    a1;
  else begin
    a2;
    c;
  end;
  var_bool := true;
  repeat
    b;
    if d then begin
      d1;
      c;
      var_bool:= false;
    end else
      var_bool:= true;
    until var_bool;
    d2;
  end.

```

(a) The text code representation of the syntax tree in Fig. 6 (b) (b) The text code representation of the syntax tree in Fig. 6 (d)

Fig. 7. The text code representation corresponding to the final syntax trees in Fig. 6 (b) and (d)

Table 1. Example cases for the complexity measuring implementation (n is the number of nodes in the flow graph and V is the complexity measure proposed in the Section 3.3). The bold line (case #3) represents the example from Figure 1.

Case #	Initial flow graph		Böhm-Jacopini method (determ.)		SCC method (non-deterministic)				
	n	V	n	V	Result count	Min V		Max V	
						n	V	n	V
1	8	3	8	3	1	8	3	8	3
2	9	9	12	4	1	12	4	12	4
3	10	9	26	11	5	17	5	32	12
4	14	36	38	18	12	25	11	63	29
5	50	156	82	64	52	71	32	82	64
6	100	276	237	154	72	112	84	289	312

transitions, respectively (as compared with 8 transitions for the first simple case). This shows that the potential advantages of the approach, in terms of graph transformations, could be applied in practice.

5 Conclusions

In this paper we take a first step towards an implementation of existing flow graph decomposition methods using graph transformations.

As stated in the introduction, well-structuredness was one of our main guidelines. We investigated several alternative and mutually complementary classical methods of flow diagram decomposition. We implemented the Böhm-Jacopini approach in terms of graph transformations employing the graph-transformation tool Groove. For the implementation we used an extended concept of equivalent flow graphs defined through the notion of context-free languages.

The Böhm-Jacopini decomposition method was enhanced and improved by using the Peterson *et al.* method that examines strongly connected components for multiple entry points and removes extra entry points by node duplicating.

In the introduction we stated that the well-structuredness of models is very important. Our full complexity measuring of a flow diagram reflects an intuitive notion of readability and enables us to compare the final syntax trees to evaluate different decomposition methods and different results of non-deterministic methods and minimize their complexity.

An important issue is to expand the set of implemented methods and apply them to improve software reliability and readability, for instance in model transformations from UMLA to Java programs. A concise review of many of other results developed in this field has been prepared in [7].

The described approach is still work in progress. The applying well-formed structures is just the first step in the general decomposition approach: the next step is to review the different cases of flow graphs with parallelism and loops and develop universal method similar simple flow graphs without parallelism.

In general, we intend to investigate the applicability of our framework to enhance a model transformation from UMLA to structured models and formally prove the correctness of this transformation. After enriching that model transformation, our long-term goal is to implement the same methods to transformations from UMLA to business process execution languages.

Acknowledgements. The research in this paper was carried out in the GRASLAND project, funded by the Dutch NWO (project number 612.063.408).

References

1. Allen, F.E.: Control Flow Analysis. In ACM Sigplan Notices (1970)
2. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. In Communications of ACM, vol. 9, no. 5, pp. 366-371 (1966)
3. Collar, E., Valerdi R.: Role of Software Readability on Software Development Cost. In 21st Forum on COCOMO and Software Cost Modeling, Herndon, VA (2006)
4. Dumas, M., ter Hofstede A.H.M.: UML Activity Diagrams as Workflow Specification Language. In Proceedings of the UML'2001 Conference, Toronto, Canada (2001)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag, Berlin, Germany (2006)
6. Engels, G., Kleppe, A.G., Rensink, A., et. al.: From UML Activities to TAAL - Towards Behavior-Preserving Model Transformations. In Proceeding of the European Conference on Model Driven Architecture (ECMDA-FA). Lecture Notes in Computer Science 5095, Springer-Verlag, Berlin, Germany, pp. 94-109 (2008)
7. Erosa, A.M., Hendren L.J.: Taming control flow: A structured approach to eliminating goto statements. In Proceedings of ICCL, Toulouse, France, pp. 229–240 (1994)
8. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.: On Structured Workflow Modelling. In Proceedings CAiSE'2000, Stockholm, Sweden, vol. 1789, pp. 431-445 (2000)
9. Kleppe, A.G., Rensink, A.: A Graph-Based Semantics for UML Class and Object Diagram. Technical Report TR-CTIT-08-06 Centre for Telematics and Information Technology, University of Twente, Enschede (2008)
10. McCabe T.: A Complexity Measure. In IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308-320 (1976)
11. Object Management Group: Abstract Syntax Tree Metamodel, Request For Proposals (RFP) (2005), <http://www.omg.org/cgi-bin/doc?admtf/05-02-02.pdf>
12. Object Management Group: Business Process Modeling Notation, V1.1. 2008, <http://www.omg.org/docs/formal/08-01-17.pdf>
13. Peterson, W.W., Kasami, T., Tokura, N.: On the capabilities of while, repeat and exit statements. In Communications of ACM, vol. 16, no. 8, pp. 503-512 (1973)
14. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In AGTIVE 2003, Springer, Heidelberg, Germany, Vol. 3062, pp. 479–485 (2004)
15. Rensink, A., Zimakova, M. Examples of Implementation in Groove (2009), http://ewi.utwente.nl/~mzimakova/bm-mds_2009
16. Zhao, W., Hauser, R., Bhattacharya, K., Bryant B.: Compiling Business Processes: Untangle Unstructured Loops in Irreducible Flow Graphs. Technical report UABCIS-TR-2005-0505-1, Birmingham, USA (2005)

Recursive Modeling for Completed Code Generation*

Selo Sulisty, Andreas Prinz

Faculty of Engineering and Science, University of Agder
 Groosveien 36, N-4876 Grimstad, Norway
 {selo.sulisty, andreas.prinz}@uia.no

Abstract. Model-Driven Development is promising to software development because it can reduce the complexity and cost of developing large software systems. The basic idea is the use of different kinds of models during the software development process, transformations between them, and automatic code generation at the end of the development. But unlike the structural parts, fully-automated code generation from the behavior parts is still hard, if it works at all, restricted to specific application areas using a domain specific language, DSL. This paper proposes an approach to model the behavior parts of a system and to embed them into the structural models. The underlying idea is recursive refinements of activity elements in an activity diagram. With this, the detail generated code depends on the depth at which the refinements are done, i.e. if the lowest level of activities is mapped into activities *executors*, the completed code can be obtained.

Keywords: MDD, business modeling, languages, and automation.

1 Introduction

Model-driven development is a kind of innovative software development that meets general industry requirements for software development, such as reduction the operating costs, reductions in time to market, and the need for open solutions. One such model-based software development approach is OMG's Model Driven Architecture, MDA [1].

Software development using OMG's MDA framework implies creation of models of the following kinds (see Figure 1): the *computation independent model* (CIM) at the business system model level, the *platform independent model* (PIM) at the information system model level, the *platform-specific model* (PSM) at the software model level, and finally, the *code* which will automatically be generated from the PSM at the software model level.

However, to date, the fully automatic generation of the code from models is still a dream and, if at all work, restricted to specific application areas by using a domain-specific language (DSL). There are two approaches to achieve an automatic code

*This work has been supported by The Research Council of Norway in the ISIS project

generation of the behavior parts of a model; bottom-up approaches that can be done by extending programming environments, and top-down approaches that can be done by extending the modeling language, i.e. UML. This article proposes a combined approach; the top-down approach by applying recursive refinements of activity elements in an activity diagram and the bottom-up approach by defining *executors* of the lowest level of activities.

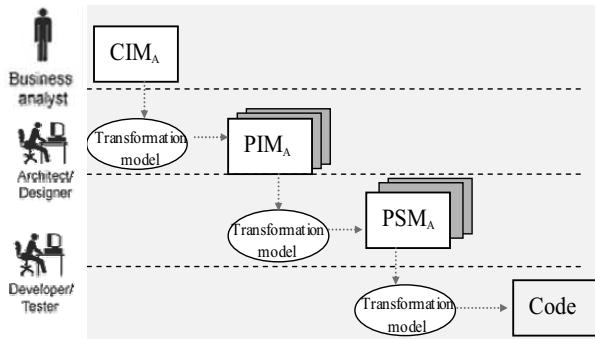


Fig.1. Software development with MDA

The reminder of the article is organized as follows: Section 2 gives a brief definition of models. Next, in Section 3 we present an overview of what modeling and programming languages are. In Section 4 we present our approach to how the code can be generated from both structure and behavior models, including an example of modeling a simple calculator application. We use UML2.1 notations and Java in the example. Section 5 is devoted to related work including the ISIS project. Finally, we draw our conclusions in Section 6.

2 Models

A model is an abstract representation of a system which describes the structure and the behavior of the system. The system itself can be anything. Models can come in many shapes, sizes, and styles, by so doing helping humans to better understand the system.

The abstraction of a system means removing the irrelevant aspects/information of a system which means that the higher abstraction level the less information a models contains. For an example, a model of a car engine gives us a better understanding of how a car engine works than the real engine installed in a car. The engine model can be very detailed with all parts of the engine (lower abstractions), or only a simple engine model describing only the functionalities (higher abstractions). In this case the simple engine model has less information than the detail.

In MDA, models can be represented in different views such as UML [6] class diagrams, activity diagrams, state machine diagrams, and collaboration diagrams. In UML, we use mainly class diagrams to model the structure part and use mainly activity diagrams, state machine diagrams and collaboration diagrams to model the behaviors' part.

2.1 Structural Models

The structure specifies what the instances of the model are; it identifies the meaningful components of the model construct and relates them to each other. The structure consists of the classes that can appear in the system as well as their mutual dependencies given as associations between the classes. The structure given by a model is used at runtime to allow the creation of several runtime objects as instances of the structure model.

As mentioned above, that in MDA, the structure is mainly modeled using UML class diagrams. The structure model in UML emphasizes the static structure of the system using objects, attributes, operations, and relationships.

2.2 Behavioral Models

The behavioral model emphasizes the dynamic behavior of a system, which can essentially be expressed in three different types of behavioral models; interaction, activity diagrams, and state machine diagrams. We use interaction diagrams to specify how the instances of model elements are to interact with each other (roles) and to identify the interfaces of the classifiers.

The UML activity diagrams are used mostly to model data/object flow systems. It is good to model the behavior of systems which;

- do not greatly depend much on external events.
- mostly having steps that run to completion, rather than being interrupted by events.
- requiring object/data flow between steps.

But since activity diagrams in UML 2.0 specified notations to communicate with external systems using *receive* and *send* signals, the activity diagram can also be used to model the systems as mentioned above (instead of a state machine diagram).

3 Modeling vs. Programming Languages

Expressing a software system by a model requires a *modeling language* as a tool. A common approach is by using *programming languages*. So, with respect to the fact that both programming languages and modeling language's ultimate aim is to produce code then we consider that they are the same. In [2], the similarities and differences between them are discussed. They have identified that modeling languages use higher abstraction, mostly using graphical representation and they are often not executable. In contrast, programming languages are executable, mostly using text representation, and of low abstraction level.

Different languages can be used to model a system at different levels of abstractions and representations, as it depends on the purpose the language. With regard to the different abstraction levels, and MOF [1], there are four different categories of language and we call it as a language model in the M2 layer (of 3+1 model layer [3]) as follows (see also Figure 2);

- business process-oriented modeling languages,
- information system- (or technology-) oriented modeling languages,
- implementation-oriented modeling languages, and
- transformation-oriented modeling languages.

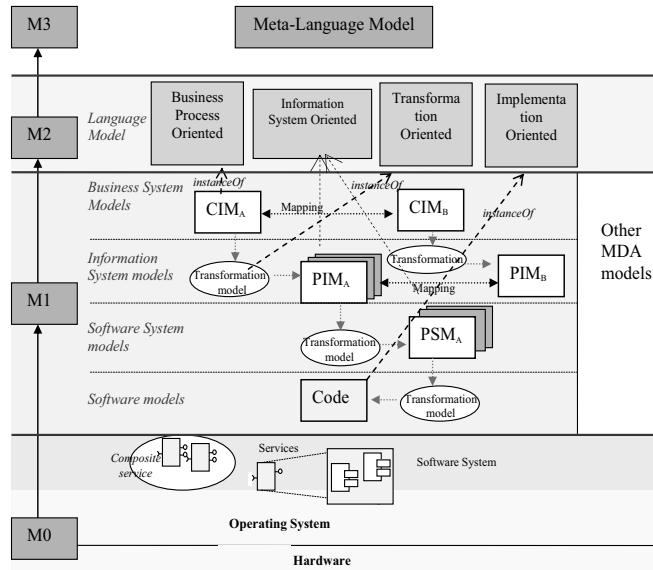


Fig.2. Models and modeling languages

All the languages mentioned above are used to generate models in the M1 layer at different levels of abstraction. Typically, implementation-oriented languages focus on the implementation of a system. The system is described in term of the implementation. These kinds of languages are what we know well as programming languages.

In software development, the main goal is always a running system. Therefore, the most important assets for the running system are the developed or generated code that is compiled and executed [4]. Whatever language we use to model a system, the main goal is, in the end (directly or indirectly), executable code. For this, a transformation-oriented language is useful. Models can be automatically transformed into a different level of abstraction (vertical transformation) and into different representation of different tools (horizontal transformation), and to code at the end.

In the context of 3+1 model layer, the code is also a model, because similar to other models in the M1 layer, the code is abstract from the real machine (hardware and operating systems). As shown in Figure 1 and 2, CIM is a model at the highest of abstraction level while the code is the model at the lowest of abstraction level. Consequently, modeling languages are also a programming language. In this way, we can consider that modeling using modeling languages is another way of (indirectly) writing code (programming), on high abstraction level as it simply concerns the focus shifting from implementation oriented languages to model oriented languages.

A completed code means that the code includes the behavioral and the structural parts. With programming languages, we write them directly, but this is not the case in modeling with modeling languages since we can use different views to represent a single system. This introduces consistency and integration problems.

In fact it is difficult to integrate different views of UML models. Several solutions for integration problems have been proposed, i.e. [15] [16]. All of them assume that automatic tools would be able to generate completed code from the integrated model. With our approach, a developer can have completed code in a way as he writes code with programming languages.

4 Writing Code using Graphical notations

To capture the behavior parts, we focus on the modeling process, as we consider this to be, in the sense of code generation, modeling using graphical-based modeling languages, i.e., UML, as simply another way of writing code. In our approach, the code is not only generated from the mapping of final PSM models, but also from all models including the PIM models, and even the CIM, models during the process of modeling.

Figure 3 shows the modeling process that is done in an interlaced (weaving) way between the structural and behavioral parts. The behavior models are used to strengthen the structural models. For example, an activity entity in an activity diagram could be transformed into an operation in a class diagram.

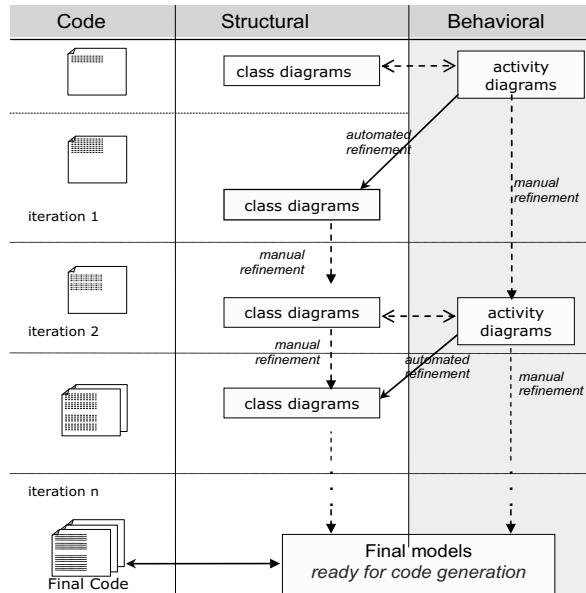


Fig.3. Weaving between structural and behavioral.

It is not fixed whether structural or behavioral part that should be modeled first. It depends on the developer's preferences. In Figure 3, we start the modeling from the structure. The structure, however, has a relation to the behavior part. When we model the behavior, implicitly, we define also partly the structure and directly also define code. As illustrated in the left most column of the figure, the code is implicitly generated when the structural and behavioral model have been defined and the lines of code is growing as the iteration increase.

The weaving from structural to behavioral and vice versa can be done many times (iteration). It depends on how complex the system is and how detail the model we want. The activities in an activity diagram should be refined into more detail with other activity diagrams, recursively. So, the more iteration (model refinements) is done, the more detail the model will be and the more code is obtained.

In this way, the activity refinements should go to more concrete and implementable (the lowest activity level). At the end, when the model will be transformed into PSM, this lowest level of activities should be mapped into available PSM components models. We call these components as activity *executors*. In this context, a model is PIM if it does not contain *executors*. The PIM can be transformed into different *executors* implemented in different programming languages.

In our approach the structure models are basically (partly) generated automatically from the behavior models, through a transformation mechanism. On the other hand, the behavior modeling must also always refer to the structural parts. In this way, the approach promotes a good traceability between the structural and behavior parts.

At the end of the refinements the final code and models that includes both of the structural and behavior parts is obtained. The models can be considered as a well formed model as it has well defined code for a specific language. At certain level of abstraction, where the model does not contain any *executors*, the code can be generated for other languages.

4.1 Transformation Rules

To support the automated refinements, we need to capture every notation in an activity diagram (behavioral) to be transformed into structural models. For this we need a set of rules.

An activity diagram consists mainly of activities, forks, decisions, and flows entities. An activity diagram can also be represented in several partitions to describe that a model of a system has several parts. All these entities (elements) can be categorized into static and dynamic entities. For example, the partition is a static entity while an activity (task) is a dynamic entity.

On the other hand, the structural part, for example a class in a class diagram, has also dynamic and static entities (elements). For example, in a class, a class name is a static entity while operations are a dynamic entity. Based on these facts, we can develop a transformation rules that transforms each entity in an activity diagram (behavioral part) into an entity in a class diagram (the structural parts). The following table shows basic rules of the transformation.

Table 1. Basic transformation rules

Activities entity (behavioral)	Structural
Partition	Class
Object (flow)	Class
Activity	Operation
Variable	Class

Of course, there are many other activity entities (elements). With this transformation rules, a developer has defined implicitly structural models when he models the behavior part.

4.2 Model to Code Transformation

The important aspect of code generation from the behavior is interpretation of every notation in an activity diagram and automatic transformation the notations into code. For this we need to define a set of transformation rules. The interpretation should be focused on the activities, control flows, and object flows.

The notion of the transformation rules follow patterns as shown in Table 2. We consider that an activity diagram can be seen as a collection of patterns. Here the patterns are used to simplify the code generation from activity diagrams.

Table 2. Patterns

Pattern	Symbol	Example in Java code
Basic	→ activity →	activity();
	X1 → [] activity →	activity(X1);
	X1 → [] activity → X2 →	X2 _x2=activity(X1);
	→ activity → X2 →	X2 _x2=activity();
Fork	X1 → [] activity → X2 →	X2=activity();
	X1 → [] activity → X2 → X3 →	if (X1) && (X2) && (X3) { activity(X1,X2,X3); }
Join	X1 → [] activity → X2 → X3 →	
Merge	[X1] ↗ ↘ activity →	if (X1 X2) { activity(); }
Decision	[X1] ↗ activityA →	if ((X0==X1) { activityA(X0); } else if ((X0==X2) { activityB(X0); }
	[X0] ↗ ↘ activityB →	
Loop	→ ↗ ↘ [X1=X2] activity → X1 ↗ ↘ [X1=X2]	while (X1!=X2) { activity(); }

The table above shows examples of possible translation code of different patterns of activity in Java. Note that the X0, X1, and X2 are objects that can be type of any.

4.3 An example

In this example we developed a simple calculator application. We explain in principle in detail how the modeling process is done and how the code can be generated automatically. We use Java code, which should be comparable for other object-oriented languages.

4.3.1 First Iteration

There is no reference whether behavior or structural should be modeled first. The only point upon it we agree is that the first model should be at a very high level of abstraction. In our approach, we first model the structural part.

Structural modeling

The calculator application consists of processor and user interface, i.e., keyboard and LCD, as shown in Figure 4, below.

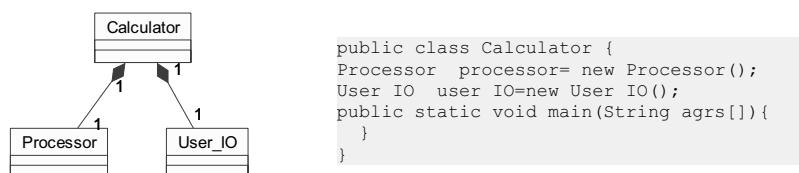


Fig.4. The Calculator class diagram

At this point, it is easy to generate code from this structural part. A possible Java code for the Calculator class is shown. In this case we assumed that the Calculator class is a main class, therefore, it includes main method.

Behavior modeling

To add more code, we need to model the behavior part. In this first iteration, we can model the calculator behavior in a high abstraction level, as for example shown in the following figure.

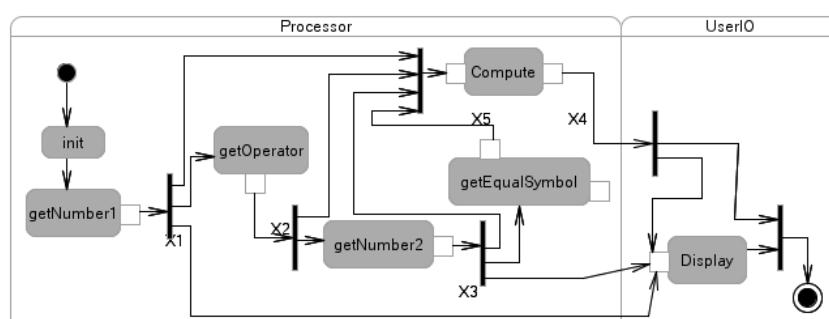


Fig.5. Activity diagram of the Calculator

To interpret the activity diagram above, we need explicit labels to indicate whether the flow is control flow or object flow. In the example we use X1, X2, X3, X4, and X5 to indicate the object flows. The following figure shows the structure part of the model as a result of automatic refinement of the activity diagram. Here, the activity diagram (behavior) is used to strengthen the structure.

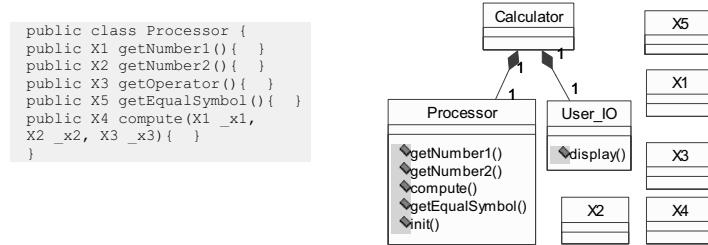


Fig.6. Refined Calculator class diagram

The basic rule is very simple (see Table 1, Section 4.1). All activities are transformed into operations of classes. The complete transformation, i.e. including the operation arguments, is presented in Table 2, Section 4.2. Based on the activity and class diagrams above, one possible generated code for the Calculator class (Figure 4) is demonstrated as follows.

```
public class Calculator {
    Processor _processor= new Processor();
    User_IO _user_IO=new User_IO();
    public X1 _x1=new X1();
    public X2 _x2=new X2();
    public X3 _x3=new X3();
    public X4 _x4=new X4();
    public X5 _x5=new X5();

    public static void main(String args[]){
        processor.init();
        x1 = processor.getNumber1();
        x1= x1;
        userIO.display( x1);

        x2 = processor.getOperator();
        x2= x2;

        x3 = processor.getNumber2();
        x3= x3;
        userIO.display( x3);

        x5 = processor.getEqualSymbol();
        x5= x5;

        if ((X1)&&(X2)&&(X3)&&(X5)){
            x4 = processor.compute( x1,  x2,  x3,  x5);
        }
        x4= x4;
        userIO.display( x4);
    }
}
```

We now have more lines of code in the Calculator class. We do not, at the moment, think about race condition, for example interpretation of flows outgoing from a fork.

4.3.2 Second iteration

At this iteration, we refine both the structure and behavior models manually. The aim is to make the model more detailed.

Structural Modeling

What we then can do is then refining the class diagram manually. One possible refinement is for instance, defining data type of X1, X2, X3, X4 classes. For example, in the class diagram we defined datatype of X2 class as follows.

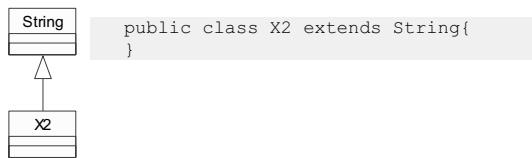


Fig.7. Data type definition

To be able to generate code automatically for a specific platform, as a final refinement, the data type definition should be defined for a specific platform as well. For example, X2 can be defined by of type java.lang.String. By using this model, the X2 class will logically have the same properties as String. However, practically speaking, it is not always true since the String class, in this case, is defined as final and therefore un-extendable. However, for this we propose the notation above as a general way of modeling the type definition in UML. It is the code generator that will determine the code resulting from the model.

In the example we define X1, X3 and X4 of type integer and X5 of type String. Other structural (class diagram) refinements that we can do are defining attributes, introducing new classes, and multiplicity, etc.

Behavioral Modeling

At the same iteration step, we refine manually the behavior part of the model. The refinement is done by defining sub activities of all activities we have defined, for example defining a sub activity of compute, shown in Figure 8. Note that not all activities notations are shown as in **add** activity.

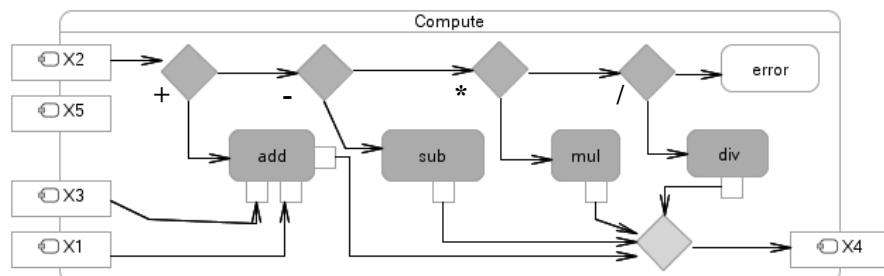


Fig.8. Sub activity diagram of *Compute* operation

From this sub activity diagram we can refine the structural model by introduce new classes, as we did in the previous iteration. In this case, the **Compute** activity is automatically be transformed into **Compute** class which has operations of **add()**, **sub()**, **mul()**, **div()**, and **error()**;

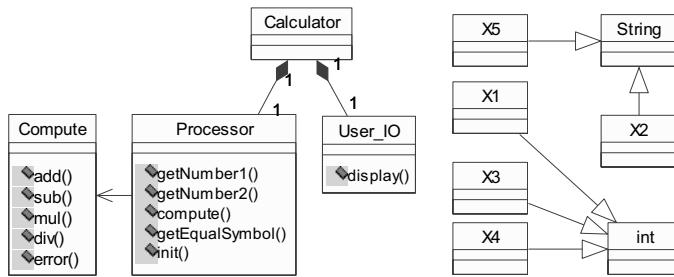


Fig.9. Refined class diagram

The following shows an example of possible generated code (expanded) for compute operation on the Processor class, based on the sub-activity we have defined above. Note that some lines (other operations) are not shown in this code.

```

public class Processor {
    .
    .
    .
    public int compute(X1 x1, X2 x2, X3 x3, X5 x5) {
        Compute _compute=new Compute();
        if (_x2.equals("+")){_x4=_compute.add(_x3 + _x1);}
        else if (_x2.equals("-")){_x4=_compute.sub(_x3- _x1);}
        else if (_x2.equals("*")){_x4=_compute.mul(_x3*_x1);}
        else if (_x2.equals("/")){_x4=_compute.div(_x3/_x1);}
        else {_compute.error();}
        return _x4;
    }
}
  
```

4.3.3 Third to nth Iteration

The more iteration and the deeper refinement of the activity are done the more code that can be obtained. However, the code is not complete enough regarding the arithmetical operations. The reason is there are no mathematical expressions in the model. To amend this situation, we propose notations for basic logical operations, basic arithmetical operations, and basic string manipulation as shown in Table 3.

The modeling process (activity modeling) goes from high abstraction level to detail. In the case of component-based development the lowest level of activities must be mapped directly into existing components and classes (models) in order to get a PSM model and code. Thus the components and classes which act as *executors* of the

activities at implementation levels should be abstracted into models. In this context, as long the activities are not mapped into executors then the model is still PIM.

Table 3. Basic operations

Notation	Category
 $X1 + X2$ $X1 * X2$ $X1 - X2$ $X1 / X2$	Basic arithmetic operations
 OR AND NOT INC DEC	Logical operations
 $X2 = X1.substring(a, b)$	String operation

See Figure 10 for a more complete behavior model, which includes the arithmetic notations. SysML (UML profile) [14] has defined a parametric to model mathematical expressions. But the expression itself uses text which might be difficult to interpret.

It is possible to refine an activity using a state machine diagram. It depends, however, on the kind of system, as explained in Section 2.2.

5 Related Works

Top-down approaches of the behavior capturing can be done either by extending the modeling languages or changing the modeling process and styles. The need for extending UML modeling languages, for example, is described in [9] [10]. Both identified that syntax and semantics are the most serious problems with the UML 2.0. There is also a problem with consistency between model views as presented in [11].

In [12], a modeling process is proposed. To obtain the final code from the behavior and the structural parts, they proposed a weaving modeling that can be done in two ways; code level weaving and model level weaving. However this weaving is completed after the models have been considered completed. In [13], a mixed graphical and text modeling where modelers can insert code manually into a graphical model, is proposed.

In our opinion, we need a modeling tool that provides assistance for an interlaced modeling process from the structural and the behavior parts vice-versa. It aims to have fully automatic code generation. The tool must also supports for mapping the activity elements into the activity *executors* (existing components).

In the Infrastructure for integrated services (ISIS) project [5] we deal with service development. Within the ISIS project, a tool called Arctis [7], which is built on UML 2.0 and is a part of the SPACE [8] approach to services (software) design, has also been developed. In Arctis, a model is described in terms of building blocks (structure) that contains activities diagrams (behavior). A building block can contain other

building blocks as parts of the internal activity diagram. We considered that a building block is an activity *executor*. Figure 11 illustrates this.

Modeling in Arctis, in principle, assumes that the activities *executors* have already defined. Therefore, we are now exploring automatic wrap to extract building blocks from existing software components in term of Arctis, in order to include these in the design of new software systems or services.

6 Conclusions

This paper presents a combined approach to generate a completed code from the structural and the behavior models. We use the top-down approach by applying recursive refinements of activity elements in an activity diagram, and use the bottom-up approach by developing a tool for the creation of activity *executors* and for the mapping mechanism of the lowest level of activities into *executors*. In our context, a model that does not include *executors* is a PIM model.

With respect to code generation, we consider that the modeling is just the focus shifting from implementation-oriented languages to graphical (model)-oriented languages. Thus, to be able to generate code, modeling languages need to abstract basic arithmetical operations, basic logical operations, and string manipulations, as the basic activity *executors*.

References

1. OMG's MDA, Architecture, <http://www.omg.org/mda>
2. Sun, Y., Zekai, D., Marjan, M., Jeff, G., and Barrett, B. : "Is My DSL a Modeling or Programming Language?", Whitepaper. labri.fr/perso/reveille/DSPD/2008/papers/4.pdf (2008)
3. Jean Bézivin.: On the unification power of models. In: Software and System Modeling, Volume 4, Number 2, May 2005. DOI: 10.1007/s10270-005-0079-0 (2005)
4. Oldevik, J. Neple, T. Aagedal, J. O. 2004. Model Abstraction versus Model to Text Transformation, White paper. University of Kent, Great Britain
5. BIP, "ISIS: Infrastructure for Integrated Services project", a research collaboration between NTNU, HIA, TellU, and Telenor Norway, 2007-2011 (2007)
6. OMG "UML 2.1 Specification", Object Management Group, 2007, <http://www.omg.org/spec/UML/2.1.2/>
7. Kraemer, F.A.: Arctis and Ramses: Tool Suites for Rapid Service. In: Proceedings of NIK-2007 (Norsk informatikkonferanse). (2007)
8. Kraemer, F.A.: Engineering Reactive Systems. Doctoral thesis at NTNU, 2008, ISBN 978-82-471-1146-8 (2008)
9. France, R. and Rumpe, B.: Model Driven Development of complex software: A research roadmap. In: Future of Software Engineering, IEEE (2007)
10. Schattkowsky, T. and Fårster, A.: On the pitfalls of UML 2 activity modeling. In :Proceedings of Workshop on Modeling in software engineering, IEEE (2007)
11. Wang, H., Feng, T., Zhang, J., and Zhang, K.: Consistency check between behavior models. In: Proceeding of ISCIT2005. (2005)

12. Mraidha, C., Gerard, S., Terrier, F., and Benzakki, J.: Two aspect approach for a clearer behavior model. In: Proceedings of the 6th International Symposium on Object-oriented real-time Distributed Computing. (2003)
13. Scheidgen, M. 2008. Textual Modelling Embedded into Graphical Modelling. In proceeding of 4th ECMDA-FA 2008, Berlin, Germany, June 9-13 (2008).
14. SysMIL, OMG SysML v. 1.1 [November 2008] <http://www.sysml.org/specs.htm> (2008)
15. Bowman, H., Steen, M., Boiten, E.A., Derrick, J.: A Formal Framework for Viewpoint Consistency. In Formal Methods in System Design, Springer Netherlands, Volume 21, Number 2 / September, 2002 (2002). DOI 10.1023/A:1016000201864
16. Chiorean, D., Pasca, M., Carcu, A., Botiza, C., Moldovan, S.: Ensuring UML models consistency using the OCL Environment. In: 6th International Conference on the Unified Modeling Language - the Language and its applications. San Francisco (2003)

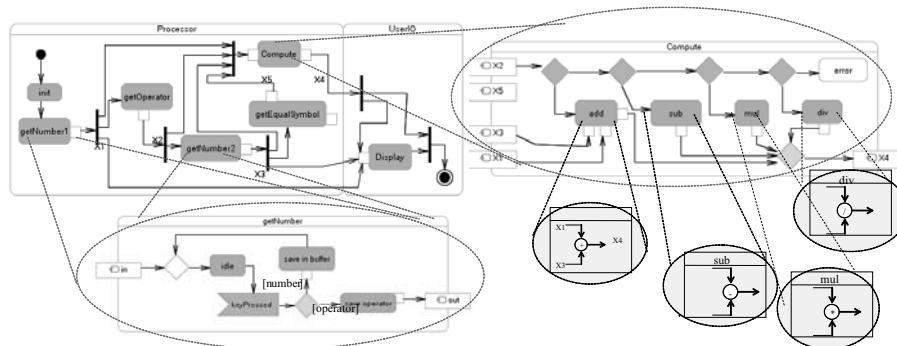


Fig 10. More completed behavior model of the Calculator

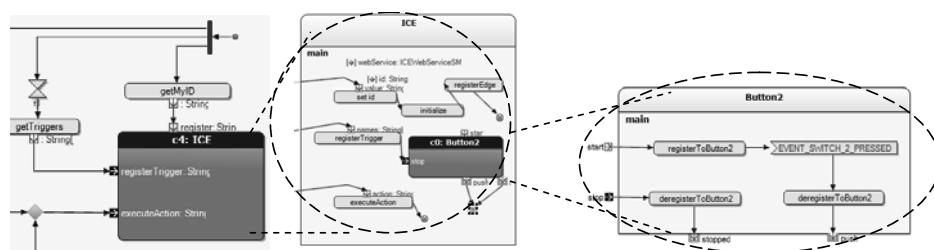


Fig 11. An example of Arctis models

Embedding Process Models in Object-Oriented Program Code

Moritz Balz and Michael Goedicke

University of Duisburg-Essen
 {moritz.balz,michael.goedicke}@s3.uni-due.de

Abstract. Process modeling has usually a strong connection with run time platforms that allow dynamic configuration and adjustment. While this is reasonable for the operation of large applications, it is of no help in cases when program code is to be engineered, documented or verified with respect to process models. We propose a design pattern for process models that allows to embed complete model semantics in object-oriented program code fragments. The program code can thus be validated and executed with respect to process model semantics and design tools.

1 Introduction

Process models can be designed, validated and executed by various tools and frameworks. The focus of these approaches is to allow dynamic configuration of processes with as little relation to source code as possible, sometimes even without involvement of IT departments. This entails the existence of frameworks that read process descriptions, walk through the process and execute business logic attached to several stages of it. The flexibility to design and alter processes descriptively comes at the cost of overhead for this frameworks and also for integration layers to the program code constituting the business logic.

While this makes sense for large distributed applications, it is of no help in cases when program code is to be engineered, documented or verified with respect to process models. Such program code is reasonable in cases when integrated or even embedded applications are developed and it is undesirable or even impossible to incorporate the overhead associated with process run times. Moreover, little support is given by current modeling technologies to design behavioral aspects of program code in all detail. Modeling languages that aim to represent all behavioral semantics tend to become as complex as general-purpose programming languages [1]. When program code is generated from models, it usually has to be refined to meet the requirements in detail. The code evolves in this case because of enhancements, corrections or tuning activities and can hardly be related back to a model afterwards [2].

So, there is in many cases a gap between abstract model definitions and program code that represents (behavioral) execution logic in detail. We earlier proposed to embed behavioral models in object-oriented code structures to maintain different levels of abstraction in the same program code [3]. This is possible when we define a design pattern that represents the complete semantics of a behavioral model. We will develop

such a pattern for process models in this contribution. The program code can thus be considered at different levels of abstraction: A process model can always be extracted from the pattern code and be used to design and validate the model in appropriate tools, for example visual editors. At the same time, the model structures are embedded in and connected with arbitrary other program code that may represent arbitrary (behavioral) semantics.

To explain the approach, this contribution is structured as follows: In section 2 we will describe the process model we want to represent by a design pattern; the pattern itself and the related program code fragments are explained in section 3. Based on this we show how process models can be designed with a modeling tool in the Eclipse IDE and executed with a lean framework afterwards in section 4. We evaluate the approach by means of a sample application in section 5, give an overview of related work in section 6 and afterwards conclude in section 7.

2 Model Definition

The objective of the pattern to develop is to engineer program code for applications whose behavior can be expressed with process semantics. We must for this purpose define a specific process model that describes the features of interest. We will stick to the process meta model defined by the Eclipse IDE's [4] Java Workflow Tooling [5] project which aims to bridge differences between existing process model notations to build a uniform editing and monitoring tool set.

The definition of this pattern as such is not innovative. But, we need a clear definition to create the design pattern afterwards that is embedded unambiguously in object-oriented program code. Additionally, the model must – since it will be expressed in program code fragments – define interfaces to arbitrary other program code that belongs to the same applications under development.

The model itself consists of two types of elements, nodes and transitions, whose properties control the flow of the process.

2.1 Activities

Activity nodes contain an activity, which means execution of arbitrary business logic of the application that contains the process model. Execution control is here passed from the scope of the process model to the application. For the definition of the design pattern this means that an interface to this arbitrary program code must be defined that will be used here.

Each activity is – as borrowed from JWT – executed by an application which is defined by a class and method name. The activity can have input and output data.

2.2 Transitions and Guards

Transitions are directed edges that connect nodes and thus guide the process flow. They are most important when they emanate from *decision nodes*. In this case they mark the begin of branches that are based on a decision depending on variables. These variables

are named and represent the state space of the application as far as it is of interest to the process model. At this point we have a second interface to the business logic of the application that provides the according variable values.

The variables are evaluated by guards that are attached to the transitions emanating from the decision node. The guards consist of an expression that considers some of the variables and produces a Boolean value. During execution, the guard of each transition is evaluated; the first transition whose guard evaluates to *true* is selected.

The expressions in guards consist of

- a left side which is always a variable identifier,
- an operator out of $\{==, !=, <, <=, >, >=\}$,
- a right side which is a literal value or another variable identifier.

These simple expressions can be aggregated to complex expressions with union ($\|$) and intersection ($\&\&$) operators.

2.3 Simple Nodes

Apart from activity and decision nodes, further nodes exist that have no properties except a name. However, they control process flow with the transitions attached to them. Beginning and termination of a process are marked by *start nodes* and *end nodes*. For concurrent execution of processes, *fork nodes* can define the beginning of a parallel execution. Concurrent flows are united in *join nodes*. *Merge nodes* bring different possible paths after decisions together.

3 Pattern Definition

Now that we have outlined the process model features that we want to use to engineer program code, we can define the design pattern that represents these model semantics. This approach builds upon the concept of so-called *internal DSLs* [6], i.e. domain-specific languages that are embedded into other languages (host languages). Semantics of DSLs are by this means available inside a general-purpose programming language. Furthermore, *attribute-enabled programming* [7] uses the capability of modern programming language versions to incorporate type-safe, compiled meta data to annotate source code fragments. These annotations can be used to make program code semantically interpretable even at run time. We combine these existing concepts to embed the model definitions in code fragments. The rules for this are explained below. For each section an illustration is given that shows the assignment of program code fragments to process model elements. They are taken from a larger example that will be explained in section 5.

The mapping has so far been implemented for the Java programming language, version 5 [8], which includes all necessary language elements, especially annotations for type-safe meta data [9].

3.1 Nodes and Transitions

The foundation of process models is the graph structure of process nodes and transitions between them. In the object-oriented program code fragments for the pattern, each process node is represented by a class definition which implements the given interface `IProcessNode` (“node class” in the following). This interface defines no methods, but allows to distinguish between node classes and arbitrary other classes type-safely. The class name corresponds to the process node name. Process nodes (except activity nodes, see below) are decorated with meta data denoting their type (one of the enumeration constants `START`, `END`, `FORK`, `JOIN`, `DECISION`, or `MERGE`).

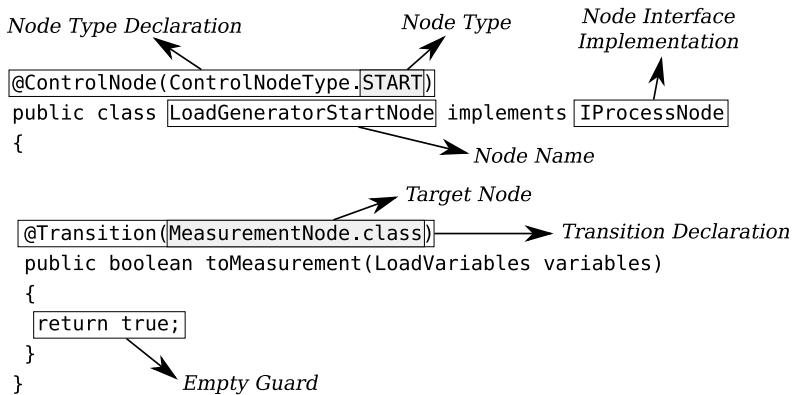


Fig. 1. A process node represented in the design pattern. The single program code fragments refer to the node’s name and type, a transition declaration and an empty guard.

Each transition is represented by a method (“transition method” in the following) inside the node class it emanates from. A transition method is decorated with meta data containing a pointer to the class definition representing the transition’s target node. With these simple definitions, the basic process graph is embedded in static object-oriented structures. The program code that is required for this is shown in figure 1.

3.2 Activities

The part of the design pattern that represents activities is slightly more complex since it considers method contents also. As defined in the model above, we will in this context need variables that are available in the model semantics and represent input and output parameters of activities. The variables are represented by an interface definition. This interface provides methods for retrieving and setting variable values (“getter” and “setter” methods in the following) which are identified by the names of these methods.

An activity node is also represented in the program code by a class definition. However, activity nodes implement an interface `IActivityNode` which itself extends `IProcessNode`. This interface defines a method `void action(Object actor, Object variables)` whose body is an activity in program code. Passed to this method are two parameters:

First, the *actor* which is a type that encapsulates business logic of the surrounding application; second, an implementation of the variables interface.

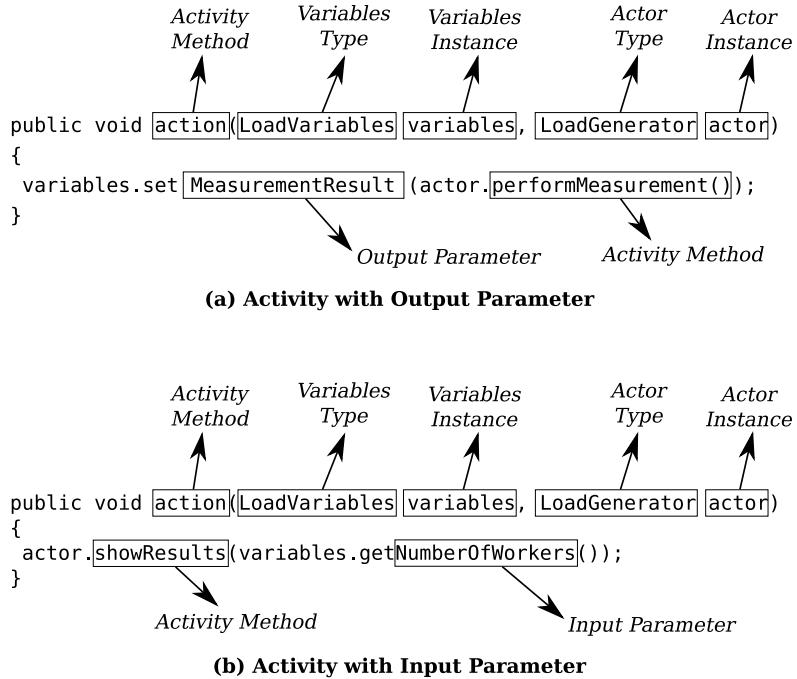


Fig. 2. Two exemplary activity methods. The first returns a value which is used as an output parameter of this activity node by calling a setter method in the variables facade. The second has an input parameter which is represented by a getter method.

The method body content follows these rules, as can be seen in figure 2:

- The actual activity is performed with a call to one of the methods of the actor instance.
- The actor method can optionally take parameters. These parameters must each be an expression containing a call to a getter method of the variables implementation. They represent input parameters to this activity as defined in the process model.
- The actor method can optionally return a value which is passed to a setter method of the variables implementation. This represents an output parameter of this activity. Since the return value is used for this purpose, we limit the number of possible output parameters in our pattern to 1.

3.3 Guards

As defined above, transitions between nodes are represented by methods. When the methods emanate from decision nodes, they are required to have guards, i.e. evalua-

tions of variable values to allow for decisions. For our pattern we define that these are represented by the method body of transitions methods. The body can therefore contain expressions as defined in section 2.2, but with calls to getter methods of the variables type. For example, the guard expression `value1>value2` would be represented in the source code as `variables.getValue1()>variables.getValue2()`. The according rules apply for interleaving of expressions. An example is shown in figure 3.

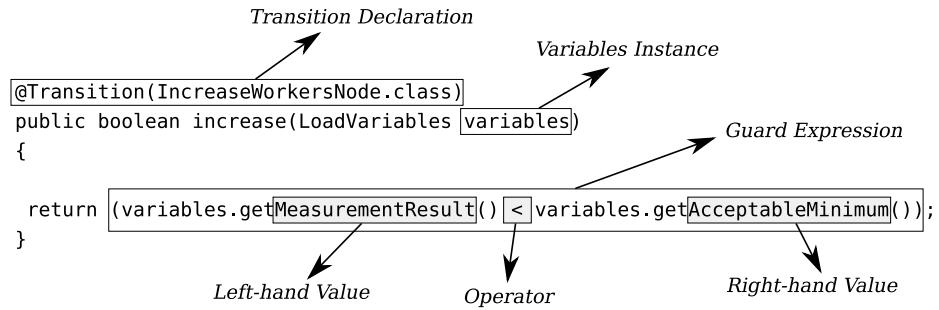


Fig. 3. A guard represented by an expression in the body of a transition method.

However, a significant part of transitions methods are defined outside of decision nodes and must have a well-defined content too. We define for this purpose that the default content of such a method is `return true`. This is a valid program code statement and will mean that no guard is defined for this transition.

4 Usage and Tools

While the use of the pattern itself may already structure the development process by facilitating the definition of a mental model by the programmer, tool support for the interpretation of the well-defined pattern fragments is desirable. We have so far developed two tools: An execution framework that interprets the fragments at run time, and a connector to a visual modeling tool that allows to design the program code with process semantics at development time.

4.1 Execution

The execution framework is based on Java reflection mechanisms. To start execution of a process from arbitrary program code, the start node, an instance of the variables type and the actor instance are passed to the framework. It instantiates all node types that are reachable by transition annotations. Afterwards the framework walks through the process by following transitions. Guards of transitions are evaluated in decision nodes by invoking the transition method and passing the variables instance to it. In activity nodes the framework calls the activity method and passes the variable and actor instance to it. In fork nodes, new threads are started that walk through the additional

defined paths. The process is continued until the current state is the end state or the framework runs into a deadlock in a decision node when no guard evaluates to *true*.

4.2 Modeling

The full benefit of working at different layers of abstraction can only be realized with modeling tools that allow for visual representation of process semantics. We chose to develop a connector to the JWT process modeling framework mentioned above. The connector is realized inside the Eclipse's Java Development Tools [10] environment. When the user selects a package with Java classes inside Eclipse, the connector allows to transform the source code inside this package into a process model. For this purpose it searches for classes that are of interest to the pattern, i.e. classes that implement the `IProcessNode` or `IActivityNode` interface. It builds a graph from them afterwards and considers the following content:

- Node classes and transition methods are directly transformed to nodes and edges in the model by using their respective names.
- Data types related to Java data types are defined for all variables.
- All methods defined in the actor class are defined as applications in the process model.
- Activity methods are analysed according to the rules defined in section 3.2.
- If a transition method is inside a decision node and contains statements other than `return true`, it is a guard and as such validated for conformity with the rules defined in section 3.3 and extracted in a tree structure of single statements linked with the valid set operators.

The model information is serialized into an XMI [11] file which is interpreted by the JWT editor. The connector interprets the pattern complete enough to extract the whole model. An example is shown in figure 4. The load generator application that can be seen there is extracted from the code of an example that will be introduced in section 5.

Currently, our connector does not create or change Java code from models designed in the editor. However, since all relevant information about the model is available in the editor, this is as straight-forward as extracting the model out of the source code.

5 Example

As can be seen in the graphical representation, these simple rules for our pattern are sufficient to embed complete process semantics in program code. We will now illustrate the code structures in detail by means of an example application written in the Java programming language. The example is a process model in a load generator application for performance tests. We assume that the load generation is a complex operation that may include networking issues, for example remote controlling of worker threads on different physical machines. The details of load generation are therefore not in the focus of the process model, but only the aspect of the behavior that controls the measurement, which can be reduced to a few well-defined variables.

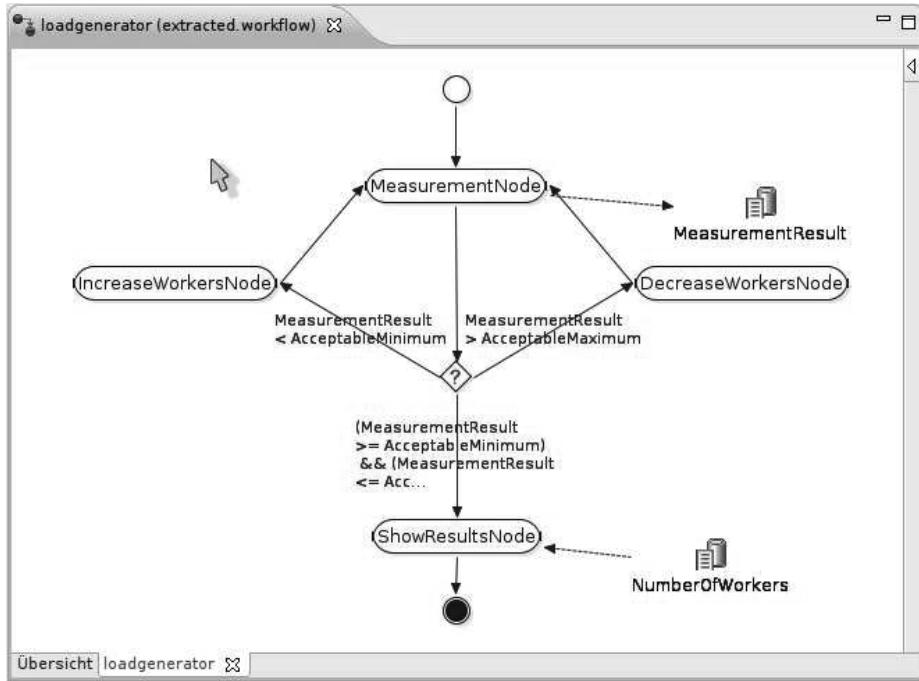


Fig. 4. A process model extracted from a pattern in the JWT editor. The example application shown here is discussed in detail in section 5.

5.1 Description

The user can give a range of acceptable response times defined by two variables *AcceptableMinimum* and *AcceptableMaximum* which denote the limits. Load is generated by a number of worker threads which is available in the variable *NumberOfWorkers*. The number of workers is adjusted depending on the measurement results across different single measurements. The latest measurement result, represented by the average response time of the system under test, is stored in the variable *MeasurementResult*. The related interface *LoadVariables* in Java is shown in listing 1.1. It contains getter methods for the range limits and the number of workers and pair of get and set method for the measurement result.

This process model controls the flow of the application, the activities are delegates to arbitrary program code. For this purpose, the actor type *LoadGenerator* is defined as an interface to the business logic as seen in listing 1.1. The most important method here is *performMeasurement* (see listing 1.2) that generates load with the given number of worker threads (initially 1). It returns a number value with the measured average response time. Afterwards, the number of workers must be increased (method *increaseWorkers*) or decreased (method *decreaseWorkers*) when the load was below or above the given range. After the measurement has finished, the result, i.e. the acceptable num-

```

public interface LoadVariables
{
    double getAcceptableMinimum();
    double getAcceptableMaximum();

    double getMeasurementResult();
    void setMeasurementResult(double result);

    int getNumberOfWorkers();
}
public class LoadGenerator
{
    public void increaseWorkers()
    { // ... }
    public void decreaseWorkers()
    { // ... }
    public float performMeasurement()
    { // ... }
    public void showResults(int numberOfWorkers)
    { // ... }
}

```

Listing 1.1. The variables interface for the load generator example with the according get and set methods

ber of workers, is shown to the user. The method *showResults* takes for this purpose one parameter with the according number of workers.

These requirements are sufficient to define the activity nodes. To complete the model, we must add the start and end node as well as a decision node that evaluates the results after each measurement. It makes the decision to increase or decrease the number of worker threads or alternatively finish the measurement based on the variable values and defines for this purpose guards as shown in listing 1.2.

5.2 Evaluation

The example shows that, based on the pattern definition, a process model can be embedded in the program code completely. The editing tool presented above allows to model the behavioral model structures visually. The relation between code and model is unambiguous, so that a transformation between both representations is possible. At the same time, the source code is the only necessary representation from which other representations for different degrees of abstraction can be extracted on demand.

In the use case of the load generator, working at different degrees of abstraction can be important. When we imagine that load generation is distributed over the network, working with time constraints for proper measurements and also using plug-ins that can generate different types of load, it would be hard to find a modeling tool that allows to express all this in behavioral models. In contrary, a framework for business processes would be a large overhead and not an appropriate solution for the need to model this simple process.

The embedded model pattern is therefore the adequate compromise: The process is embedded in arbitrary business logic with arbitrary behavior. The program semantics related to the process model are represented distinctly, the arbitrary program state is

```

public class MeasurementNode implements IActivityNode<LoadVariables, LoadGenerator>
{
    public void action(LoadVariables variables, LoadGenerator actor)
    {
        variables.setMeasurementResult(actor.performMeasurement());
    }

    @Transition(EvaluateMeasurementNode.class)
    public boolean toSetResults(LoadVariables variables)
    {
        return true;
    }

    @ControlNode(ControlNodeType.DECISION)
    public class EvaluateMeasurementNode implements IPProcessNode
    {
        @Transition(IncreaseWorkersNode.class)
        public boolean increase(LoadVariables variables)
        {
            return (variables.getMeasurementResult() < variables.getAcceptableMinimum());
        }

        @Transition(DecreaseWorkersNode.class)
        public boolean decrease(LoadVariables variables)
        {
            return (variables.getMeasurementResult() > variables.getAcceptableMaximum());
        }

        @Transition>ShowResultsNode.class)
        public boolean showResults(LoadVariables variables)
        {
            return ((variables.getMeasurementResult() >= variables.getAcceptableMinimum())
                && (variables.getMeasurementResult() <= variables.getAcceptableMaximum()));
        }
    }
}

```

Listing 1.2. The nodes that initiate and evaluate a measurement. The type is specified with annotations and implemented interfaces; The action method contains the activity with an output parameter, the other methods are transitions with guards

reduced to well-defined variables for this purpose. The definition of the pattern code structures is certainly a little overhead, but allows to visualize the model on demand and therefore comprehend the concept behind code structures easily. For the definition in the context of the example application, the embedded model is therefore capable of providing different levels of abstraction with respect to process semantics.

6 Related Work

Many approaches exist that try to engineer program code at different layers of abstraction.

Model Round-Trip Engineering concepts [12] allow to abstract from source code by synchronizing it to abstract models. However, they require manual effort [13] and are therefore not unambiguous enough to create these abstractions ad-hoc. In this context,

the attribute-oriented programming approach has already been explored to map UML models to code structures [14], similar to Framework Specific Modeling Languages [15]. In contrast to embedded models, these approaches rely on the existence of different representations for different abstraction levels, and do not avoid round trip engineering.

Specification languages like the Java Modeling Language (JML) [16, 17] or the introspection capabilities of Smalltalk [18] enable the definition of semantic information and modeling constraints inside object-oriented source code and provide an extensive syntax for this purpose. These meta data are though not related to formal models like processes and are not detailed enough to extract model definitions completely into abstract representations.

Contrary to the model checking approach of Java PathFinder [19] we do not consider the semantics of a complete program as such. The same applies to the concept of Introspective Model-Driven Development [20] that aims to identify unknown model structures in the source code. Instead, we relate only selected parts of it that are well-defined beforehand to existing formal models.

Similar to run time systems that execute process models are Executable Models, for example “executable UML” [21], that aim to avoid working with source code completely by direct execution of model specifications. This relies on the assumption that entire applications can be expressed as models, which is – especially for behavioral modeling – not realistic from our point of view, as mentioned in the introduction.

7 Conclusion

We presented an alternative approach to behavioral modeling. To allow working at different levels of abstraction when program code is engineered, we defined a pattern that represents process model semantics in object-oriented source code fragments. Arbitrary source code can thus be enriched with semantical information where applicable. Although the pattern definition is still elementary so far, we have already proven that the approach is capable to fulfill the requirements: The creation of the connector to the JWP process modeling tool shows that an unambiguous interpretation of the pattern fragments is possible. We can also execute the process model at run time while it is still part of the program code of arbitrary applications.

Future work will focus on the aspect of editing the source code: Instead of only extracting the model from the code, we will create a connector for the opposite direction that creates source code from a model or merges changes back to the source code. Our vision is a modeling tool that allows for continuous working at different abstraction levels by offering different views on the same program code to engineer.

References

1. Fowler, M.: PlatformIndependentMalapropism (2003) <http://martinfowler.com/bliki/PlatformIndependentMalapropism.html>.
2. Brown, A.W., Iyengar, S., Johnston, S.: A Rational approach to model-driven development. IBM Systems Journal **45**(3) (2006) 463–480

3. Balz, M., Striewe, M., Goedicke, M.: Embedding Behavioral Models into Object-Oriented Source Code. In: Software Engineering 2009. Fachtagung des GI-Fachbereichs Softwaretechnik, 2.-6.3.2009 in Kaiserslautern. (2009)
4. The Eclipse Foundation: ECLIPSE website <http://www.eclipse.org/>.
5. The Eclipse Foundation: JWT website <http://www.eclipse.org/jwt/>.
6. Fowler, M.: InternalDslStyle (2006) <http://www.martinfowler.com/bliki/InternalDslStyle.html>.
7. Schwarz, D.: Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. ON-Java.com (June 2004) <http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html>.
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: JavaTM Language Specification, The 3rd Edition. Addison-Wesley Professional (2005)
9. Sun Microsystems, Inc.: JSR 175: A Metadata Facility for the JavaTM Programming Language (2004) <http://jcp.org/en/jsr/detail?id=175>.
10. The Eclipse Foundation: Eclipse Java Development Tools (2008) <http://www.eclipse.org/jdt/>.
11. OMG: MOF 2.0 / XML Metadata Interchange (XMI), v2.1.1 specification (2007)
12. Sendall, S., Küster, J.: Taming Model Round-Trip Engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development. (2004)
13. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal **45**(3) (2006) 451–461
14. Wada, H., Suzuki, J.: Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In Briand, L.C., Williams, C., eds.: Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings. Volume 3713 of Lecture Notes in Computer Science., Springer (2005) 584–600
15. Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with Round-Trip Engineering. [22] 692–706
16. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-Oriented Software. The KeY Approach. Springer-Verlag New York, Inc. (2007)
17. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. In Kilov, H., Rumpe, B., Simmonds, I., eds.: Behavioral Specifications of Businesses and Systems, Kluwer (1999) 175–188
18. Ducasse, S., Gîrba, T.: Using Smalltalk as a Reflective Executable Meta-language. [22] 604–618
19. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal **10**(2) (2003)
20. Büchner, T., Matthes, F.: Introspective Model-Driven Development. In: Software Architecture, Third European Workshop, EWSA 2006, Nantes, France, September 4-5, 2006. Volume 4344 of Lecture Notes in Computer Science., Springer (2006) 33–49
21. Mellor, S.J., Balcer, M.J.: Executable UML. Addison-Wesley (2002)
22. Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings. Volume 4199 of Lecture Notes in Computer Science., Springer (2006)

