Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools

Holger Giese[1], Tihamér Levendovszky[2], and Hans Vangheluwe[3]

[1] Department of Computer Science
University of Paderborn
D-33098 Paderborn, Germany
hg@upb.de

[2]Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
tihamer@aut.bme.hu

[3]Modelling, Simulation & Design Lab
School of Computer Science
McGill University
Montréal, Québec, Canada
hv@cs.mcgill.ca

Abstract. This paper reports on the findings of the first Workshop on Multi-Paradigm Modeling: Concepts and Tools. It contains an overview of the presented papers and of the results of three working groups which addressed multiple views, abstraction, and evolution. Besides this, a definition of the problem space, the main concepts, and an appropriate terminology for multi-paradigm modeling as presented and discussed during the workshop are provided.

Keywords Modeling, Meta-modeling, Multi-Paradgim Modeling, Multi-Formalism.

1 Introduction

Complex software-based systems today often integrate different beforehand isolated subsystems. Thus, for their model-driven development multiple formalism at different levels of abstraction from possibly different domains have to be integrated. This is especially true when besides general purpose languages such as the UML also domain specific languages are employed. In this first workshop on Multi-Paradigm Modeling (MPM) at the MoDELS conference, a forum for researchers and practitioners to discuss the resulting problems and challenges has been set up.

An initial invited talk was given by Hans Vangheluwe in order to provide some generally agreed upon definitions of multi-paradigm modeling.

The paper continues with a definition of the problem space, main concepts, and terminology for multi-paradigm modeling in Section 2. Then, the presented papers are located within the introduced problem space in Section 3 before we summarized the findings of the working groups which have been set up within the workshop in Section 4. Finally, a list of the program committee follows in Section 5.

2 Multi-Paradigm Modeling

In this section, the foundations of Multi-Paradigm Modeling (MPM) are presented. In particular, we introduce *meta-modeling* and *model transformation* as enablers for Multi-Paradigm Modeling. MPM encompasses both *multi-formalism* and *multi-abstraction* modeling of complex systems. To provide a framework for the above, the notion of a *modeling language* is first dissected. This leads quite naturally to the concept of meta-modeling as well as to the explicit modeling of model transformations. The notion of abstraction is explored in the working group results section 4.2.

Models are an *abstraction* of reality. The structure and behavior of systems that we wish to analyze or design can be represented by models. These models, at various *levels of abstraction*, are always described in some *formalism* or *modeling language*. To "model" modeling languages and ultimately synthesize visual modeling environments for those languages, we will break down a modeling language into its basic constituents [1]. The two main aspects of a model are its syntax (how it is represented) on the one hand and its semantics (what it means) on the other hand.

The syntax of modeling languages is traditionally partitioned into *concrete syntax* and *abstract syntax*. In textual languages for example, the concrete syntax is made up of sequences of *characters* taken from an *alphabet*. These characters are typically grouped into *words* or *tokens*. Certain sequences of words or *sentences* are considered valid (*i.e.,* belong to the language). The (possibly infinite) *set* of all valid sentences is said to make up the language. Costagliola et. al. [2] present a framework of visual language classes in which the analogy between textual and visual characters, words, and sentences becomes apparent. Visual languages are those languages whose concrete syntax is visual (graphical, geometrical, topological, . . .) as opposed to textual.

For practical reasons, models are often stripped of irrelevant concrete syntax information during syntax checking. This results in an "abstract" representation which captures the "essence" of the model. This is called the *abstract syntax*. Obviously, a single abstract syntax may be represented using multiple concrete syntaxes. In programming language compilers, abstract syntax of models (due to the nature of programs) is typically represented in *Abstract Syntax Trees* (ASTs). In the context of general modeling, where models are often graph-like, this representation can be generalized to *Abstract Syntax Graphs* (ASGs).

Once the syntactic correctness of a model has been established, its meaning must be specified. This meaning must be *unique* and *precise*. Meaning can be expressed by specifying a *semantic mapping function* which maps every model in a language onto an element in a *semantic domain*. For example, the meaning of a Causal Block Diagram can be specified by mapping onto an Ordinary Differential Equation. For practical reasons, semantic mapping is usually applied to the abstract rather than to the concrete syntax of a model. Note that the semantic domain is a modeling language in its own right which needs to be properly modeled (and so on, recursively). In practice, the semantic mapping function maps abstract syntax onto abstract syntax.

To continue the introduction of meta-modeling and model transformation concepts, languages will explictly be represented as (possibly infinite) sets as shown in Figure 1. In the figure, insideness denotes the sub-set relationship. The

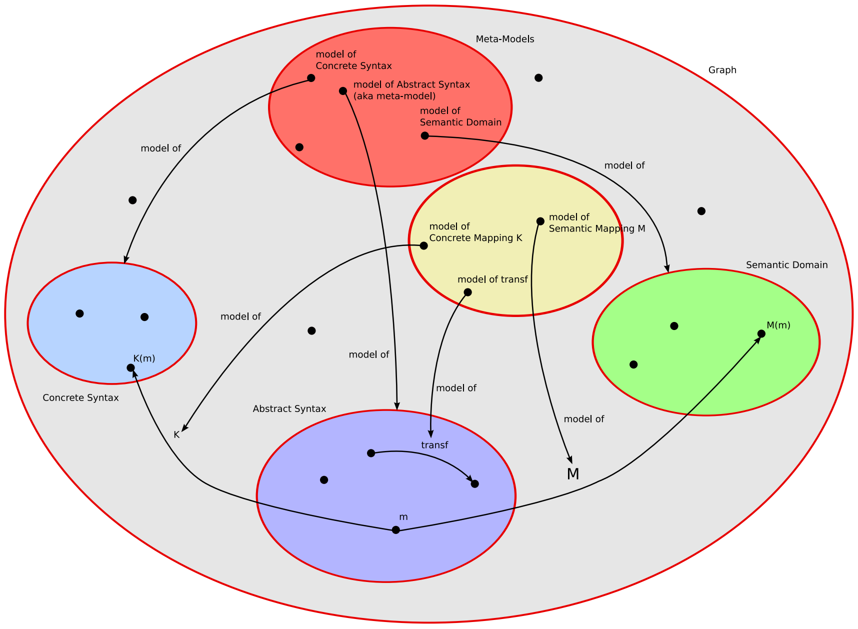

Fig. 1. Modeling Languages as Sets

dots represent model which are elements of the encompassing set(s).

As one can always, at some level of abstraction, represent a model as a graph structure, all models are shown as elements of the set of all graphs Graph. Though this restriction is not necessary, it is commonly used as it allows for the design, implementation and bootstrapping of (meta-)modeling environments. As such, any modeling language becomes a (possibly infinite) set of graphs. In the bottom centre of Figure 1 is the abstract syntax set A. It is a set of models stripped of their concrete syntax.

Meta-modeling is a heavily over-used term. Here, we will use it to denote the explicit description (in the form of a finite model in an appropriate meta-modeling language) of the Abstract Syntax set. Often, meta-modeling also covers a model of the concrete syntax. Semantics is, however, not covered. In the figure, the Abstract Syntax set is described by means of its meta-model. On the one hand, a meta-model can be used to *check* whether a general model (a graph) *belongs to* the Abstract Syntax set. On the other hand, one could, at least in principle, use a meta-model to *generate* all elements of the language. This explains why the term meta-model and grammar are often used inter-changeably.

Several languages are suitable to describe meta-models in. Two approaches are in common use:

1. A meta-model is a *type-graph*. Elements of the language described by the meta-model are instance graphs. There must be a *morphism* between an instance-graph (model) and a type-graph (meta-model) for the model to be in the language. Commonly used meta-modeling languages are Entity Relationship Diagrams (ERDs) and Class Diagrams (adding inheritance to ERDs). The expressive power of this approach is often not sufficient and a *constraint language* (such as the Object Constraint Language) specifying constraints over instances is used to further constrain the set of valid models in a language. This is the approach used by the OMG to specify the abstract syntax of the Unified Modeling Language (UML).
2. A more general approach specifies a meta-model as a transformation (in an appropriate formalism such as Graph Grammars) which, when applied to a model, verifies its membership of a formalism by *reduction*. This is similar to the syntax checking based on (context-free) grammars used in programming language compiler compilers. Note how this approach can be used to model type inferencing and other more sophisticated checks.

Both types of meta-models can be *interpreted* (for flexibility and dynamic modification) or *compiled* (for performance).

Note that when meta-modeling is used to synthesize interactive, possibly visual modeling environments, we need to model *when* to check whether a model belongs to a language. In *free-hand* modeling, checking is only done when explicitly requested. This means that it is possible to create, during modeling, syntactically incorrect models. In *syntax-directed* modeling, syntactic constraints are enforced at all times during editing to prevent a user from creating syntactically incorrect models. Note how the latter approach, though possibly more efficient, due to its incremental nature –of construction and consequently of checking– may render certain valid models in in the modeling language unreachable through incremental construction. Typically, syntax-directed modeling environments will be able to give suggestions to modelers whenever choices with a finite number of options present themselves.

The advantages of meta-modeling are numerous. Firstly, an *explicit* model of a modeling language can serve as *documentation* and as *specification*. Such a specification can be the basis for the *analysis* of properties of models in the language. From the meta-model, a modeling environment may be *automatically* generated. The flexibility of the approach is tremendous: new languages can be designed by simply *modifying* parts of a meta-model. As this modification is explicitly applied to models, the relationship between different variants of a modeling language is apparent. Above all, with an appropriate meta-modeling tool, modifying a meta-model and subsequently generating a possibly visual modeling tool is orders of magnitude *faster* than developing such a tool by hand. The tool synthesis is *repeatable* and *less error-prone* than hand-crafting.

As a meta-model is a model in an appropriate modeling language in its own right, one should be able to meta-model that language's abstract syntax too. Such a model of a meta-modeling language is called a *meta-meta-model*. It is noted that the notion of "meta-" is relative. In principle, one could continue the meta- hierarchy ad infinitum. Luckily, some modeling languages can be meta-modeled by means of a model in the language itself. This is called *meta-circularity* and it allows modeling tool and language compiler builders to *bootstrap* their systems.

A model in the Abstract Syntax set (see Figure 1) needs at least one concrete syntax. This implies that a concrete syntax mapping function κ is needed. κ maps an abstract syntax graph onto a concrete syntax model. Such a model could be textual (*e.g.,* an element of the set of all Strings), or visual (*e.g.,* an element of the set of all the 2D vector drawings). Note that the set of concrete models can be modeled in its own right. Also, concrete syntax sets will typically be re-used for different languages. Often, multiple concrete syntaxes will be defined for a single abstract syntax, depending on the user. If exchange between modeling tools is intended, an XML-based textual syntax is often used. If in such an exchange, space and performance is an issue, an binary format may be used instead. When the formalism is graph-like as in the case of a circuit diagram, a visual concrete syntax is often used for human consumption. The concrete syntax of complex languages is however rarely entirely visual. When, for example, equations need to be represented, a textual concrete syntax is more appropriate.

Finally, a model m in the Abstract Syntax set (see Figure 1) needs a unique and precise meaning. As previously discussed, this is achieved by providing a Semantic Domain and a semantic mapping function M. This mapping can be given informally in English, pragmatically with code or formally with model transformations. Natural languages are ambiguous and not very useful since they cannot be executed. Code is executable, but it is often hard to understand, analyze and maintain. It can be very hard to understand, manage and derive properties from code. This is why formalisms such as Graph Grammars are often used to specify semantic mapping functions in particular and model transformations in general. Graph Grammars are a visual formalism for specifying transformations. Graph Grammars are formally defined and at a higher level than code. Complex behavior can be expressed very intuitively with a few graphical rules. Furthermore, Graph Grammar models can be analyzed and executed. As efficient execution may be an issue, Graph Grammars can often be seen as an executable specification for manual coding. As such, they can be used to automatically generate transformation unit tests.

Within the context of Multi-Paradigm Modeling, we have chosen to use the following terminology.

- A *language* is the set of abstract syntax models. No meaning is given to these models.
- A *concrete language* comprises both the abstract syntax and a concrete syntax mapping function κ. Obviously, a single language may have several concrete languages associated with it.
- A *formalism* consists of a language, a semantic domain and a semantic mapping function giving meaning to model in the language.
- A *concrete formalism* comprises a formalism together with a concrete syntax mapping function.

This terminology will be used in the sequel.

3 Presented Papers

The paper [3] summarizes the main achievements with respect to Mechatronic UML and relates it to Multi-Paradigm Modeling. The approach combines control engineering, electrical engineering, mechanical engineering, and software engineering disciplines to describe and verify reconfigurable mechatronic systems. The multidisciplinary nature of Mechatronic UML gives a good case study for multiparadigm modeling: different parts of a mechatronic system are described by different formalisms, such as differential equations or timed automata.

The paper [4] presents a tool named Computer Aided Method Engineering (CAME). This approach uses hierarchical activity diagram to model an arbitrary software development process. To these process steps, models can be attached. The modeling languages are created with metamodeling techniques. The models created for different paradigms are assembled manually.

The popularity of block diagrams motivates the work [5], which offers a translational semantics for block diagrams by syntactically translating them into Haskell. The declarative notion of Haskell facilitates more rigorous specification as opposed to its imperative counterparts, such as C. The translation applies syntactic Haskell extensions developed by the authors.

The paper [6] uses an approach underpinned by abstract algebraic and categorical constructs. The main idea is to formalize the semantics by specifying the domains as lattices of coalgebras. Between the lattices, Galois connections can be established. If this connection is maintained during the abstractions or concretizations, the important properties are preserved. In order to check the consistency of distinct domains, pullback constructs are provided to derive a common specification. These results can be applied to formalize the composition of multi-paradigm applications.

The paper [7] proposes a formalism for modeling language composition with a low-level language. The low-level language referred to as L3 consists of three aspects: structural, descriptive and behavioral. The multi-paradigm composition technique is illustrated with two simplified UML diagrams, namely, the class and activity diagrams enhanced with OCL constraints.

The paper [8], which can be found in this volume, discusses a conceptual approach to define declarative model integration languages. The integration behavior is bound to the metamodel. Furthermore, the authors build a conceptual framework which realizes the complex integration operations on the global level to efficient and simple local operators.

4 Working Group Results

4.1 Multiple Views

The first working group addressed the topic of multi-view modelling. Multi-view modelling is concerned with the common practice of modelling a single system by means of a collection of view models. Each of these view models can possibly be represented in a different concrete formalisms. As discussed in Section 2, differences between concrete formalisms may be at the level of concrete syntax, abstract syntax, or even semantics. Together, the multiple views allow a modeller to express all relevant knowledge about a system under study. Allowing multiple views in multiple concrete formalisms allows the modeller to express different aspects of his knowledge in the most appropriate fashion, thereby minimizing accidental complexity.

Multi-view modelling does come at a price though. The different views should be consistent. In particular, if one view is modified, other views describing the same aspect of the system may need to be updated. Note that updating may be trivial if the views only differ in concrete syntax. In the worst case however, the semantics of the different formalisms in which the views are expressed may differ. In this case, formalism transformation may be required. It is noted that updating (in a Model-View-Controller fashion) is in principle always possible if update mappings are available between all views. For efficiency reasons, the quadratic (in the number of views) number of required mappings and the quadratic (in the number of view models) number of updates can be reduced to a linear number if it is possible to describe a single repository model of which all views are projections.

Also, one often needs to know whether a collection of views completely describes a system (given some notion of completeness). The issues mentioned above are exacerbated if different views describe the system at different levels of abstraction. The working group discussed abstraction at length and came to similar conclusions as those of the second working group (though not formalized). Hence, we refer to the next section for a treatment of this subject.

Jean-Marie Favre pointed out the existence of a mega-model of multi-view modelling in the reverse engineering community. This mega-model relates *views* which need to conform to *viewpoints*. Those in turn are used to cover *concerns*. Each of these may be described in an appropriate formalism.

4.2 Abstraction

The second working group worked on the topic of abstraction and how models of the same and different type (formalism) are related to each other during the model-driven development using abstraction and its opposite refinement in different forms.

As foundation for the notion of abstraction, the group started with defining the *information* contained in a model M as the different questions (properties) $P = I(M)$ which can be asked concerning the model ($|P|$ and $p, p' \in P : p \neq p'$) and either result in true or false ($M \models p$ or $M \not\models p$).

For a model, it holds in general that only a restricted set of questions (properties) are correctly addressed by the model w.r.t. the original matter. Thus, for example, questions concerning the color of a diagram or the layout of a text do not matter. These relevant questions (properties) and the related notion of a bit, then served also to define abstraction as well as several related relations.

A relation between two models M_1 and M_2 can have the character of an *abstraction*, *refinement*, or *equivalence* relative to a non empty set of questions (properties) P.

– In case of an *equivalence*, we require that for all $p \in P$ holds: $M_1 \models p \iff M_2 \models p$. We write $M_1 =_P M_2$.
– If M_1 is an *abstraction* of M_2 with respect to P it holds for all $p \in P$ holds: $M_1 \models p \Rightarrow M_2 \models p$. We write $M_1 \sqsupseteq_P M_2$.
– We further say that M_1 is a *refinement* of M_2 iff M_1 is an *abstraction* of M_2. We write $M_1 \sqsubseteq_P M_2$.

We also have a second case of abstraction and refinement when only comparing the scope given by the set of questions (properties) considered in two models M_1 and M_2:

– We have an *equivalent scope* if $I(M_1) = I(M_2)$. We write $M_1 =_I M_2$.
– We have a more *abstract* scope if $I(M_1) \subseteq I(M_2)$. We write $M_1 \sqsupseteq_I M_2$.
– We further say that M_1 has a *refined* scope of M_2 iff M_1 has an *abstracted scope* of M_2. We write $M_1 \sqsubseteq_I M_2$.

The group then employed this definition to describe the role of abstraction and refinement for some general development steps:

In case of a *analysis model*, a more abstract model M_a is derived from the concrete model M in order to prove or disprove that a certain set of properties P holds. If the abstract model provides all required information concerning P ($I(M_a) \supseteq P$) we can distinguish the case that (1) both models are equivalent ($M_a =_P M$) or M_a is an abstraction of M ($M_a \sqsupseteq_P M$):

$$(1)\quad \forall p \in P : M_a \models p \iff M \models p \qquad (2)\quad \forall p \in P : M_a \models p \Rightarrow M \models p.$$

These facts can be used to transfer the fulfilment of p from M_a to M. Note that usually the verification or analysis of p is only feasible for M_a. The equivalence or abstraction between the models is then used to propagate the result for p.

While in case of equivalence the full result can be propagated, for abstraction the check $M_a \models p$ is only sufficient to conclude $M \models p$. The propagation is not valid for $\neg p$ as there is $M_a \models \neg p$ is not necessary for $M \models \neg p$.

A typical development step in computer science is *model refinement*: A refined model M_2 is derived from the abstract model M_1 by adding details to the model. The considered set of properties P can be either fixed or extended in the refinement step ($I(M_2) \supseteq I(M_1) = P$). Due to the definition of refinement for $M_2 \sqsubseteq_P M_1$ holds: $\forall p \in P : M_1 \models p \Rightarrow M_2 \models p$.

During the development the check $M_1 \models p$ is then used to determine that any refinement step preserves this property. Thus, we can characterize the strategy as a pessimistic *risk elimination* step which excludes solutions if it is not guaranteed that for all its valid implementations (refinements) also p must hold.

While refinement is common in computer science, in engineering and related disciplines the typical development step is *approximation* which is rather different. Approximation can be seen as refinement with respect to negated properties: $\forall \neg p \in P : M_1 \models \neg p \Rightarrow M_2 \models \neg p$.[1] This effectively means that approximation is an optimistic approach which only eliminates *impossible solutions*. If a property p has already been falsified for M_1 ($N_1 \models \neg p$), we refuse all solution M_2 which cannot fulfill p.

4.3 Model Evolution

One of the main problems for a wide scale acceptance of model engineering practices in industry is the lack or the immaturity of methods and tools that allow to confidently switch to a fully model driven software development process. In conventional software development, for instance, source code versioning systems are commonplace, whereas it is still largely unclear of how adequate versioning should be applied in a model driven context.

Another pressing problem that was the topic of group discussion is the evolution of metamodels representing the abstract syntax of modeling languages. Such an evolution would alter the metamodel and therefore possibly render all models conforming to the original metamodel obsolete.

Hence, support for migrating models from the original to the changed metamodels ought to be provided. Ideally, this would come in the form of transformations that could migrate models towards newer versions of metamodels. Such transformations could possibly be derived automatically.

It is still an open question how the actual evolution of metamodels could be carried out. Perhaps it is feasible to find certain recurring "evolution patterns" similar to refactoring operations, which would ease the derivation of migrating transformations. A second possibility would be to allow "free-hand editing" of metamodels, in which case tool support should allow to at least partially load models into newer versions of metamodels and - for further manual editing - provide a comprehensive list of model elements that do not match the new

[1] In practice, M_1 is usually an idealization w,r,t. p where an approximation is only extremely likely.

metamodel. In both cases, it is advisable to store traceability information, for instance to be able to provide backwards compatibility.

Apart from discussing these more technical challenges that call for tool support, the discussion elaborated on what kinds of metamodel evolution there are, and what the needs for evolution might be.

We could identify two basic kinds of evolutions. The first would be a purely *syntactic evolution*, which would result in adding "syntactic sugar" to the metamodel, for purposes of making the modeling language more convenient to use and comprehend. One example would be to introduce model elements, that represent structures built of more basic model elements. As an example, the Business Process Execution Language (BPEL) offers convenient constructs such as Flow or Sequence, which could alternatively be modeled by linking up activities accordingly on a fine-grained level. Models expressed in either way, however, have the same semantics.

The second kind of evolution would be *semantic evolution*, where the semantics of the model elements are changed or new elements are introduced whose semantics have to be determined. This can take place through changing a metamodel and according to that changing its semantic mapping towards a semantic domain. An explicit mapping towards a semantic domain, however, does often not exist, but a code generator or interpreter is employed to make models executable. Changes to the generator would represent a change in the semantics of the language. Essentially this poses a challenge for appropriate configuration management to bind metamodels, models and their respective generators.

The purpose of such syntactic evolution could be to enhance the learnability or usability of a modeling language, whereas semantic evolution would go towards enhancing the appropriateness and expressivity of a modeling language.

The discussion concluded with the understanding that metamodel evolution should not simply be about providing means to arbitrarily alter metamodels, but be a way to continuously maintain the quality of metamodels by ensuring their fitness for task.

This would possibly require metrics for measuring the quality of metamodels and the appropriateness of the expressivity or usability of the respective modeling languages. Such metrics would indicate when a modeling language ought to actually undergo evolution, to avoid "uncontrolled" modifications that may introduce ambiguities or distort the understandability and hence the practical applicability of a modeling language.

5 Program Committee

Michael von der Beeck *BMW (DE)*
Jean Bézivin *Université de Nantes (FR)*
Heiko Dörr *DaimlerChrysler AG (DE)*
Jean-Marie Favre
Institut d'Informatique et Mathématiques Appliquées de Grenoble (FR)
Reiko Heckel *University of Leicester (UK)*

Jozef Hooman *University of Nijmwegen (NL)*
Gabor Karsai *Vanderbilt University (US)*
Anneke Kleppe *University of Twente (NL)*
Ingolf H. Krüger *University of California, San Diego (US)*
Thomas Kühne *Technical University Darmstadt (DE)*
Juan de Lara *Universidad Autónoma de Madrid (ES)*
Jie Liu *Microsoft Research (US)*
Mark Minas *University of the Federal Armed Forces (DE)*
Oliver Niggemann *dSPACE GmbH (DE)*
Pieter Mosterman *The MathWorks (US)*
Bernhard Schätz *TU Munich (DE)*
Andy Schürr *Technical University Darmstadt (DE)*
Hans Vangheluwe *McGill University (CA)*
Bernhard Westfechtel *University of Bayreuth (DE)*

References

1. Harel, D., Rumpe, B.: Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Jerusalem, Israel (2000)
2. Costagliola, G., Lucia, A.D., Orefice, S., Polese, G.: A classification framework to support the design of visual languages. J. Vis. Lang. Comput. **13** (2002) 573–600
3. Henkler, S., Hirsch, M.: A multi-paradigm modeling approach for reconfigurable mechatronic systems. Technical report, Budapest University of Technology and Economics, Dept. of Automation and Applied Informatics, Genova, Italy (2006)
4. Saeki, M., Kaiya, H.: Constructing multi-paradigm modeling methods based on method assembly. Technical report, Budapest University of Technology and Economics, Dept. of Automation and Applied Informatics, Genova, Italy (2006)
5. Denckla, B., Mosterman, P.J.: Block diagrams as a syntactic extension to haskell. Technical report, Budapest University of Technology and Economics, Dept. of Automation and Applied Informatics, Genova, Italy (2006)
6. Streb, J., Alexander, P.: Using a lattice of coalgebras for heterogeneous model composition. Technical report, Budapest University of Technology and Economics, Dept. of Automation and Applied Informatics, Genova, Italy (2006)
7. Braatz, B.: An integration concept for complex modelling techniques. Technical report, Budapest University of Technology and Economics, Dept. of Automation and Applied Informatics, Genova, Italy (2006)
8. Reiter, T., Kepler, J., Retschitzegger, W., Altmanninger, K.: Think global, act local: Implementing model management with domain-specific integration languages. In: Lecture Notes in Computer Science, Satellite Events at the MoDELS 2006 Conference, Genova, Italy (2006)