Electronic Communications of the EASST
Volume 21 (2009)

Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM 2009)

Systematic Transformation Development

Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer

10 pages

Guest Editors: T. Levendovszky, L. Lengyel, G. Karsai, C. Hardebolle
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Systematic Transformation Development

Thomas Kühne[1], Gergely Mezei[2], Eugene Syriani[3], Hans Vangheluwe[3], and Manuel Wimmer[4]

[1] Victoria University of Wellington tk@ecs.victoria.ac.nz
[2] Budapest University of Technology and Economics gmezei@aut.bme.hu
[3] McGill University {esyria,hv}@cs.mcgill.ca
[4] Vienna University of Technology wimmer@big.tuwien.ac.at

Abstract: Despite the pivotal significance of transformations for model-driven approaches, there have not been any attempts to explicitly model transformation languages yet although a number of benefits are to be gained. First, transformation developers may change the design of their transformation languages by modeling, rather than programming. Second, they may use environments to create transformations that are customized with respect to the input and output languages involved. In this paper, we use a running example to identify, discuss, and demonstrate some of the above advantages. In particular, we explore and suggest ways to systematically support developers in creating transformation languages by means of semi-automated metamodeling.

Keywords: model transformation, metamodeling, domain-specific transformation

1 Introduction

Model-driven approaches are gaining popularity both in the form of being based on standard modeling languages, such as the UML, as well as domain-specific modeling languages. In both instances, the aim is to increase developer productivity, in the case of the former by raising the level of abstraction at which systems can be specified and in the case of the latter by lowering the impedance mismatch between a modeling language and its application domain.

There are still many open problems with respect to the economic development of domain-specific modeling languages, but their definition is well understood. This shifts the focus on transformations which have a number of applications among which are: (1) establishing transformation chains from high-level to low-level specifications, (2) providing semantics for a source language by mapping it to a target language, and (3) creating a consistent mapping between two or more models. A number of transformation paradigms exists, e.g., template-based, rule-based, triple graph grammars, with or without explicit control flow [CH06]. They are supported by various implementations such as ATL [JK06], ATOM^3 [LV02], GREAT [AKK+06], QVT [Obj08], VMTS [LLMC05]. They provide tremendous value for developers, but in each implementation the transformation paradigm is hard-coded to be used as is. The implementations do not provide a way to interrogate or modify transformation definitions as first-class transformation models.

This is surprising as there are a number of benefits to be gained when treating transformations as first-class citizens which are explicitly modeled and amenable to introspection and modification. We identify the following potential advantages. It becomes easier to explore the language

design space by making alterations to the control flow, mapping, and pattern specification parts of the language. Obviously, this requires modeling the respective semantics, but once available, alterations to the syntax and semantics definitions of such transformation (meta-)models should be easier to perform than the respective changes in a code base. Instead of using a generic pattern specification language to be used for all input and output languages, one can utilize customized pattern specification languages on a case-by-case basis. Automating the creation of such customized pattern specification languages opens up a cost-neutral way to achieve customized transformation definition environments providing increased rigor.

In the following, we first introduce our running example which we use as the basis of our subsequent discussions. In Section 3, we investigate the automated construction of customized pattern specification languages, using the components relaxation, augmentation, and modification, exploring and discussing alternative solutions. This provides a systematic *procedure* for explicitly modeling transformation languages. Finally, we discuss related work in Section 4.

2 A Typical Transformation

The example that we will use in the remainder of the paper to illustrate our arguments is a typical case of a domain-specific language being assigned a semantics by translating it into a target formalism with known semantics. In order to define the semantics of statecharts and/or perform reachability analyses on them, one can translate them to Petri nets [LV02]. Another reason for considering this particular translation is that one can use Petri nets as a common semantic domain for statecharts, sequence diagrams, and activity diagrams. For the purposes of this paper, however, we restrict ourselves to translating finite state automata, rather than statecharts, into Petri nets. The resulting transformation definitions of this translation are much simpler but still rich enough to illustrate our arguments. Figure 1 shows both metamodels.

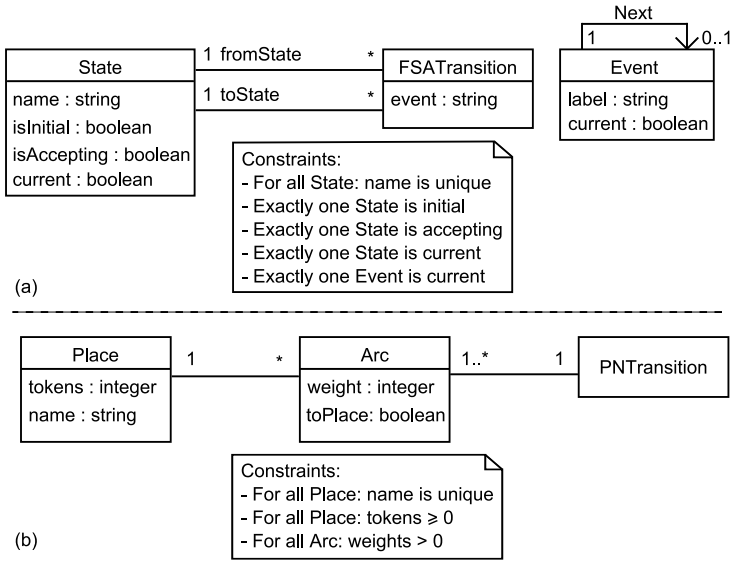

Figure 1: (a) FSA & (b) Petri Net Metamodels

2.1 Finite State Automata as Language Recognizers

We interpret our sate automata to be language recognizers, i.e., they either accept input sequences as belonging to respective regular languages or not. The top part of Figure 2 shows a sample input sequence ("yees") and a finite state automaton accepting the language y(e)*s.

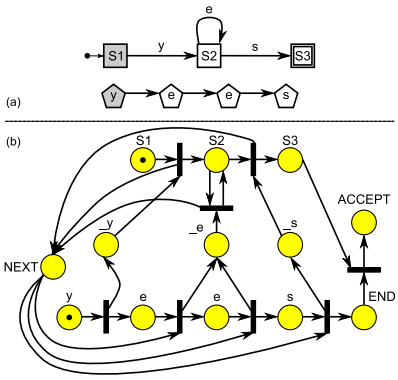

Figure 2: (a) FSA to (b) Petri Net Mapping

In our example, we want to simulate the execution of the finite state automaton in the context of receiving the events from the input sequence in order to ascertain whether the input sequence is a sentence of the language. To this end, we translate such scenarios into corresponding Petri nets (see bottom part of Figure 2).

2.2 Translating Finite State Automata To Petri Nets

Figure 3 shows an excerpt of the transformation rules that are required to translate a finite state automaton plus an input sequence into a Petri net that can be used to simulate the automaton execution. In particular, Figure 3 shows a subset of the rules that translate finite automaton states into Petri net places.

The rest of the rules dealing with the translation are similar to those shown and are not of further significance for the purposes of this paper. Note that we are using ATOM^3 [LV02] and MOTIF [SV07] and thus use concrete syntax for describing single push-out graph transformation rules, employing numerical labels to indicate identity of elements. We are well aware of the tension between the "must transition" and "may fire" semantics of finite state automata and Petri nets, respectively. In timed Petri nets, this difference may lead to a situation where a finite state automaton does not change states anymore even though it *should*, just because the Petri net used for simulating it does not fire transitions anymore, even though it *could*. However, the place/transition nets we assume do not create this mismatch and a simulator for them will fire enabled transitions.

Next, we describe and discuss the explicit modeling of transformation definitions as an enabler of customized transformation development environments.

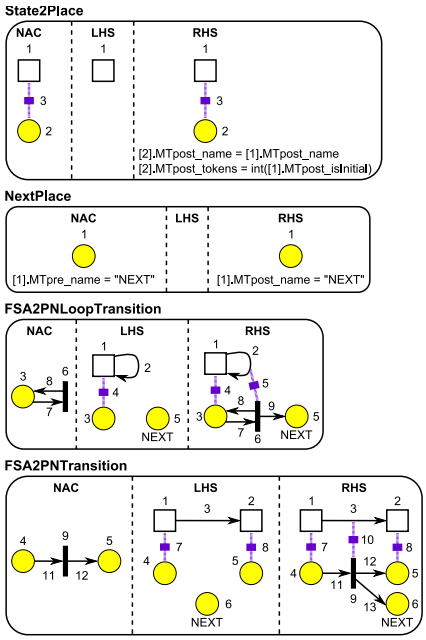

Figure 3: Translating States to Places

3 Explicit Transformation Modeling

Metamodeling[1], i.e., the explicit specification of a language's well-formedness constraints, has become popular because of a number of associated advantages: (1) the specification is not hidden in the code of a tool, making it easier to understand and correct, (2) the specification can be altered by users of the tool instead of requiring a new tool release, and (3) one can reason about the specification and the models it describes. The same advantages apply if metamodeling is not only applied to modeling language definitions, but also to transformation definitions. While

[1] Linguistic metamodeling, to be precise.

there is a considerable initial investment to be made in explicitly modeling a transformation language including its semantics, the prospect to more easily experiment with language features, customize them for certain purposes, and allow transformations to be reasoned about and/or modified makes that investment worthwhile.

Clearly, in order to enable the last aspect mentioned above, the transformation language's mapping approach, e.g., rule-based graph transformation, needs to be explicitly modeled.

Unlike the mapping and control aspects of a transformation language, its pattern specification sublanguage depends on other languages. The input and output languages of a transformation determine which pattern specifications for left-hand side (LHS) and right-hand side (RHS) can be considered well-formed. The underlying assumption here is that the pattern specification language should not be generic to fit all possible input and output languages, but specifically tailored to the input and output languages involved.

3.1 Generic versus Customized Pattern Specification Languages

The most economic approach to providing a pattern specification language is to offer a generic one. Most tools do not use concrete syntax for specifying transformation patterns and thus are able to use the same generic (often UML object-diagram-inspired) pattern specification syntax for all possible input/output languages. They often also have an underlying generic (often MOF-like) representation format which can be used to represent elements from any input/output language.

There are good reasons, however, to consider using a pattern specification language which is customized to the input/output languages involved:

- One may use pattern specification visualizations which are adapted to the languages involved. Even if no concrete syntax is used, one may still want to customize the syntax, e.g., to adequately visualize connector elements.
- A customized syntax allows excluding patterns from being specified that do not have a chance of matching subgraphs in the host graphs. For instance, in the context of Petri nets, a pattern consisting of an arc linking two places will never be matched on any valid Petri net instance (i.e., conforming to the meta-model in Figure 1).

A generic pattern specification language will allow any pattern to be expressed whether or not it will be able to match subgraphs from the input language(s) or generate subgraphs conforming to the metamodel(s) of the output language(s). Just as a plain domain-specific modeling tool has advantages for its users, guiding them to produce meaningful models, a customized transformation pattern specification tool also aids in avoiding meaningless pattern specifications.

Whether this customization is achieved by changing the representation format for each generated transformation definition environment or by just exchanging a language definition against which generic pattern specifications are checked is immaterial to the user, but a tool builder decision. In the following, we assume that, in one way or another, pattern specifications can be checked for conformance to a pattern specification language definition. As a result, a method needs to be identified that enables these conformance checks in an economic manner, while offering the transformation language user maximum benefits.

3.2 Metamodels versus Conformance Checks

Unfortunately, providing a customized pattern specification language is not as easy as simply reusing the corresponding input/output metamodels. First, demanding a full adherence of pattern specifications to original language definitions is not practical. If all minimal multiplicity requirements of language definitions were enforced, one could not specify useful patterns which refer to model fragments, ignoring minimal multiplicity requirements. Second, one may want to provide several levels of rigor with respect to checking the well-formedness of pattern specifications. While the transformation designer edits a pattern specification, one most certainly does not want to enforce all well-formedness constraints. It also should be possible to save ill-formed sketches to be worked on later. This does not mean, however, that the complete absence of all potential well-formedness checks is always the best choice in such cases. Table 1 lists potentially useful levels of conformance checking rigorousness. There are two ways to enable the use of such levels of conformance: (1) either one creates modified language definitions and performs a normal conformance check against them, or (2) one uses original language definitions, but accordingly modified conformance checks. The second option has a number of advantages:

- one can simply use the original language definitions; there is no need to create multiple variants of them.
- switching between conformance levels does not require the switch of a metamodel; the latter is quite feasible though with an appropriate architecture.
- the alternative (1. above) cannot use a standard conformance check anyhow (see Section 3.3 and Section 4).

However, there are also a number of disadvantages:

- some generic way to extend languages defined by metamodels is required; pattern specification languages require additional features beyond the original input/output languages (see Section 3.3). Customized metamodels can easily incorporate these.
- custom conformance checks are harder to reason about than custom metamodels; in the absence of a fully modeled action language, conformance checks will be implemented in some programming language making it harder to see and analyze what relation they actually implement.

Level of rigor	**Description**
Free form	no constraints at all
Valid elements	elements are typed by the metamodel
Valid multiplicities	(relaxed) multiplicity constraints are enforced
Valid constraints	(a subset of) metamodel constraints are enforced

Table 1: Levels of Conformance

- conformance checks are harder to customize by users; transformation designers can be expected to alter the transformations that yield tailored metamodels but may not be able to re-program conformance checks.

- swapping conformance checks means that the transformation development will remain the same; swapping metamodels opens up the possibility to use them for the automated generation of dedicated development environments with differing sets of control elements.

Finally, there is another motivation for supporting more than one mode of well-formedness checking which can only be enabled by using multiple metamodel versions: Typically, transformation definitions comprise layers of rules in the sense that one will expect all rules from one layer to have matched, and then match no more, before the next layer of rules will be used. This layering often exists independently of whether or not it is dealt with explicitly. In particular with in-place transformations, the input and output languages change from layer to layer. The first layer's input language is the source language while its output, the input to the next layer, will typically contain generic links which are not part of the source language (see Section 3.3). The last layer's output language is the target language, whereas all preceding layers will produce either augmented versions of it or mixtures between the source and target languages. The availability of a series of adapted metamodels may aid the transformation developer to understand what the layers involved are and assign rules to them accordingly.

We have not yet pursued the idea of using a series of transformation layer interface language definitions and it would be challenging to automate the generation of these intermediate language definitions. Luckily, however, automating the creation of customized pattern specification languages from original input/output language definitions can be automated very well.

3.3 Semi-Automated Metamodeling

The previous section motivated the use of variants of original metamodels for defining the well-formedness of pattern specifications. In this section, we discuss how one can create such variants systematically and thus automate the process.

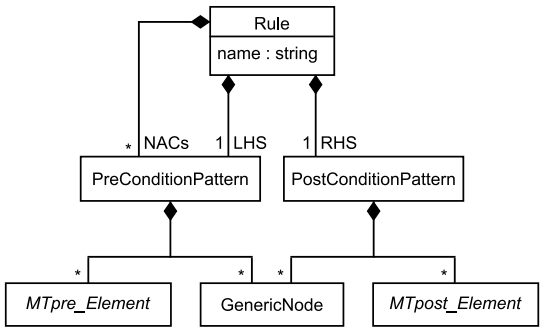

Figure 4: Rule Metamodel

Figure 4 depicts how rules refer to precondition and postcondition patterns and the pattern element they contain. When adapting transformation languages to specific input and output languages, one needs to tailor these precondition and postcondition patterns so that they are fit to be used for the respective input and output languages. We obtain the required tailored pattern specification metamodels by starting with the original language metamodels and then subjecting them to a number of changes. The required metamodel metamorphosis has three distinct components: relaxation, augmentation, and modification. Figure 5 shows an excerpt of the result of applying these steps to the finite state machine metamodel of Figure 1.

3.3.1 Relaxation

Original language definitions cannot be used as is for defining the well-formedness of pattern specifications.
First, often transformation designers aim to match for any one-of-many element types, e.g. one-of-many "connection" kinds.
Such generalizations are typically present in original language definitions but as abstract concepts which cannot be instantiated. One relaxation step therefore is to turn such abstract concepts into concrete ones.

Second, as mentioned before, enforcing minimal multiplicity constraints would be completely impractical.

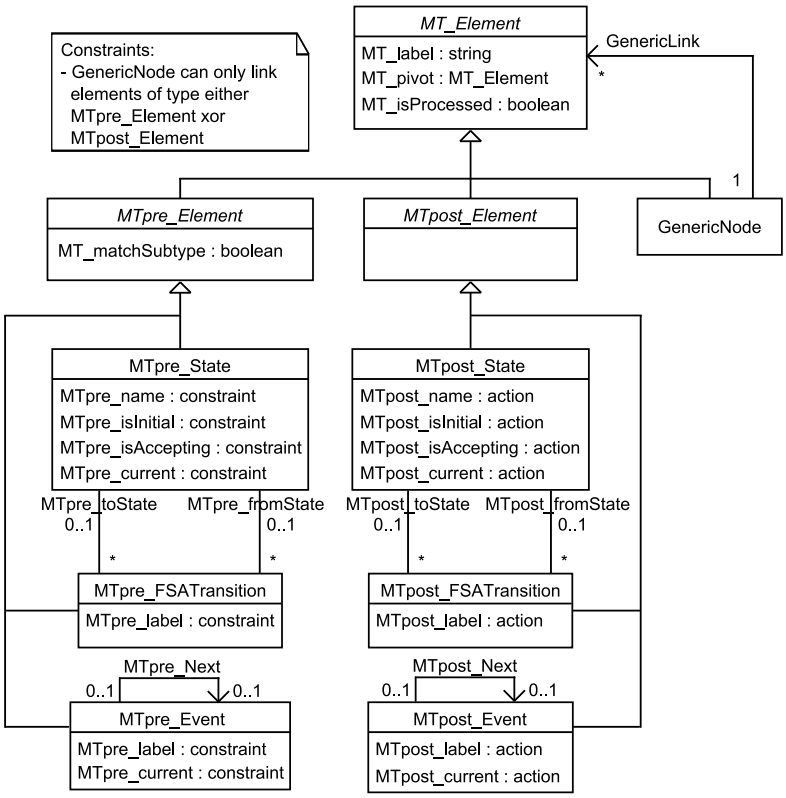

Figure 5: Generated Pattern Specification Metamodel

A further relaxation step is, therefore, to reduce all minimal multiplicities to zero (see Figure 5 for the relaxation of State multiplicities and Section 4 for a more elaborate discussion).

Third, only a subset of explicitly formulated original constraints (e.g., using OCL) can be active for the purpose of checking pattern specification well-formedness. All constraints concerned with ensuring completeness of models are potentially unsuitable for the inherent fragment-like nature of specification patterns. The relaxation process could automatically filter out constraints with the help of a corresponding naming scheme for constraints or manually provided augmentations, but we currently believe any further automation will be difficult to achieve. This is why we refer to the metamodel generation as *semi*-automated.

A potential further relaxation is to raise all maximum multiplicities to "unbounded" in order to allow intermediate results that can be helpful to drive the transformation process, despite the fact that they would be ill-formed as end results. However, we argue that purposefully violating well-formedness requirements in this way amounts to "hacking" and should be avoided. We recommend using so-called generic links for these purposes instead.

3.3.2 Augmentation

To be fit as pattern specification metamodels, input/output metamodels also need to be augmented with features required for transformation purposes.

In Figure 5, all types are made descendants of *MT_Element* so that they inherit features that all

elements that may appear in a pattern specification must have, e.g., a way to label them for identity matching. The generated metamodels also feature additional generic nodes and links which are often necessary to drive the transformation (e.g., see the generic connectors between states and places in Figure 3). Elements which are used in negative application conditions (NACs) or the LHS of rules (subtypes of *MTpre_element*) also need a flag feature that tells the pattern matcher whether to look for exact types or allow subtype matching as well. The remaining differences between the original and generated metamodel elements are all modifications of existing features.

3.3.3 Modification

The modifications that need to be applied to original metamodel elements depend on whether we want to obtain precondition (i.e., NAC and LHS) or postcondition (i.e., RHS) pattern specifications.

For precondition pattern specifications we need to replace the respective types of attributes to the type "constraint". This allows the transformation designer to specify constraints for element features, such as MTpre_name="NEXT" (see Figure 3). For postcondition pattern specification we need to allow actions rather than constraints, so that the transformation designer can set values of attributes, among other potential actions. In rule NextPlace of Figure 3, the "=" in the RHS part of the rule is an assignment action rather than an equality check. Note that the same naming and modification scheme is applied to classes, associations, and role names.

Finally, we sometimes need to modify the concrete syntax of language elements whose size or natural layout is not conducive for specifying patterns. Also, elements which are normally not rendered at all, such as instances of formerly abstract classes or association ends, need to be assigned some concrete syntax so that they may be referred to in a visual manner.

We have implemented a prototype of this procedure. A new metamodel is created as partly shown in Figure 5. In the relaxation step, we did not consider the (OCL) constraints of the respective metamodels yet and they thus have been maintained. In the augmentation step, the first two levels of the inheritance hierarchy of Figure 5 correspond to concepts from the meta-metamodel of ATOM^3/MOTIF. Finally in the modification step, our prototype did not take into account issues related to layout in the concrete syntax of the pattern elements.

Summarizing, this section has discussed various alternatives for enabling transformation designers to make use of customized pattern specification languages and environments. We proposed the semi-automated generation of customized metamodels based on the components of relaxation, augmentation, and modification.

4 Related Work

Bézivin et al. explicitly model transformations with "transformation models" [BBG+06] but for capturing the relations maintained by transformations rather than supporting their customization or generation.

The need to relax conformance rules occurs in other areas as well. Morin et al. also relax an original metamodel in order to allow the formulation of pointcut specifications in the context of aspect-oriented modeling [MBJR07]. Levendovszky et al. capture domain-specific design

patterns which also inherently are fragments of proper models [LLM09]. Instead of creating a relaxed version of the metamodel, they use relaxed conformance, i.e., "relaxed instantiation". This allows them to use one original language definition to check both proper models and design patterns. Since they only need to support this one variant of conformance checking, this is a viable approach. In general, however, the explicit modeling of transformations may require a multitude of conformance levels, making the relaxation of metamodels a more attractive option (see Section 3.2). Levendovsky et al., furthermore, observe that simply setting all minimal multiplicities to zero will allow the formulation of fragments which cannot be completed to proper models. They suggest detecting such fragments by using constraint solving. This approach is applicable in our context as well and could be realized by adding corresponding constraints to the relaxed metamodels.

5 Conclusion

Although we discussed our work and developed our artifacts in the context of ATOM^3/MOTIF, our ideas and results are by no means confined to the specifics of this combination. Our proposal to explicitly model transformation definitions is applicable to a wide range of transformation approaches.

While it is not necessary to explicitly model *all* aspects of transformation definitions, we have illustrated that there are benefits associated with each such step. First, the explicit modeling of pattern specifications allowed the semi-automatic generation of customized pattern specification language definitions based on the components of relaxation, augmentation, and modification. It thus provided a cost-effective way to obtain customized transformation development environments. In contrast to ATL higher-order transformations, ours can be fully checked for well-formedness violations. Second, the explicit modeling of transformation control structures allowed the modular addition of new behavior, such as source-level animation. Summarizing, we demonstrated the benefits of explicitly modeling transformations and proposed ways to economically enable their definition.

For future work, we would like to investigate how explicitly modeling transformation languages allows support for higher-order transformation, as treated as models, transformations can be themselves subject to transformation.

Bibliography

[AKK+06] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, A. Vizhanyo. The Design of a Language for Model Transformations. *SoSym* 5(3):261–288, September 2006.

[BBG+06] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, A. Lindow. Model transformations? Transformation models! In *MoDELS'06*. LNCS 4199, pp. 440–453. Springer Verlag, Genova (Italy), 2006.

[CH06] K. Czarnecki, S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal, special issue on Model-Driven Software Development* 45(3):621–645, July 2006.

[JK06] F. Jouault, I. Kurtev. Transforming Models with ATL. In *MTiP'05*. LNCS 3844, pp. 128–138. Springer-Verlag, January 2006.

[LLM09] T. Levendovszky, L. Lengyel, T. Mészáros. Supporting domain-specific model patterns with metamodeling. *Software and Systems Modeling* Theme Issue on Metamodeling, to appear 2009.

[LLMC05] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf. Control Flow Support in Metamodel-Based Model Transformation Frameworks. In *EUROCON'05*. Pp. 595–598. IEEE, Belgrade (Serbia), November 2005.

[LV02] J. de Lara, H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In Kutsche and Weber (eds.), *FASE'02*. LNCS 2306, pp. 174–188. Springer-Verlag, Grenoble (France), April 2002.

[MBJR07] B. Morin, O. Barais, J.-M. Jézéquel, R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *Models and Aspects workshop, at ECOOP'07*. Berlin (Germany), July 2007.

[Obj08] Object Management Group. Meta Object Facility 2.0 Query/View/Transformation Specification. April 2008.

[SV07] E. Syriani, H. Vangheluwe. Programmed Graph Rewriting with DEVS. In Nagl and Schürr (eds.), *AGTIVE 2007*. LNCS 5088, pp. 136–152. Springer-Verlag, Kassel (Germany), 2007.