Formal Support for Model Driven Development with Graph Transformation Techniques

Juan de Lara
Escuela Politécnica Superior
Universidad Autónoma
Madrid, Spain
jdelara@uam.es

Esther Guerra
Dept. Informática
Universidad Carlos III
Madrid, Spain
eguerra@inf.uc3m.es

Abstract

In this paper we give an overview of our approach for Model Driven Development (MDD), based on graph transformation techniques. In MDD, models are the primary assets in the development process. They are not only used for documentation, but also for analysis, simulation, code and test cases generation. Thus, model transformation becomes a central activity. As models can be formally described as attributed, typed graphs, we can use formal graph transformation techniques for their manipulation. In this paper, we give an overview of the different kinds of model transformation and suitable graph transformation techniques. Moreover, graph transformation can be combined with meta-modelling for further expressivity. Some of these techniques have been recently implemented in the Meta-modelling tool $AToM^3$. We use the tool to introduce an example in the component-based modelling and simulation area.

1 Introduction

In Model Driven Development (MDD) models play a central role. This is a natural evolution of the software development techniques, started more than 50 years ago. From assembler, to procedural languages, to object oriented approaches, software engineers have sought higher levels of abstraction to increase quality and productivity. In this way, the developer deals with less accidental details, concentrating in the main problems. Models are (sometimes partial) high-level representations of systems, built using a (usually visual) language. While the syntax of the languages is usually well-defined – often using meta-modelling – semantics are sometimes specified using natural language, complicating model analysis.

Several manipulations are useful in the context of MDD [7]. We classify them using two orthogonal characteristics: depending if the model is translated into a different formalism or not, and on the abstraction levels of source and target models. For expressing these manipulations, one has several options. In this work we propose using a declarative, visual and formal approach: graph transformation [4]. This is a technique based on rules, each having graphs in their left (LHS) and right hand sides (RHS). We present several variants and analysis techniques and explore their usability for MDD. Moreover, their combination with meta-modelling [1] becomes crucial in this context.

As an example of the presented concepts, we show how they have been unified in the meta-modelling $AToM^3$ tool [11], together with an example in the component-based modelling and simulation area. Thus, the main contributions of this paper are, on the one hand, the unification of different graph transformation variants in a unique framework, and their implementation in $AToM^3$. On the other hand, we have extended the component framework presented in [13], with the addition of multi-

ple views, hierarchical components and refinement.

The paper has been structured as follows. Section 2 proposes a taxonomy of model transformation. Section 3 introduces several graph transformation variants together with useful analysis techniques for MDD. Section 4 introduces MiCo, a Minimal Component visual language for modelling and simulation. Section 5 presents the implementation of the graph transformation concepts in AToM^3, together with the definition of the MiCo language. Section 6 compares with related research. Finally, section 7 ends with the conclusions and future work.

2 Model Transformation

In the context of MDD, model transformation can be classified using two orthogonal characteristics, regarding the source and target formalisms, and the abstraction level. If we look at the first characteristic, then we have transformations that are:

- **Inter-Formalism** (also known as *exogenous* [15]), which translate the model into a different formalism. This kind of transformations includes data-base or language version migration. They can also be used for analysis purposes, if the target formalism has analysis methods that can be used to study properties of the original model. In this case, the transformation should preserve the characteristics under investigation.

  Simulation Transformations are a special case. Simulation can be considered an iterative transformation from the source formalism into the "traces" formalism. A trace is an ordered sequence of events, containing information about the state variables of the simulated system, as well as the simulated time. Sometimes visual simulations do not generate traces, but perform an animation of the model. Still in this case, one is interested in the simulation yield, which includes the model state at each meaningful instant of time.

- **Intra-formalism** (also known as *endogenous* [15]), in which both source and target models are expressed in the same formalism. These transformations can be used for optimization, simplification, refactoring, etc.

If we look at the abstraction level, then we may have transformations that are:

- **Horizontal**, which convert the model into another one at the same level of abstraction. In the context of MDA, they implement PIM-to-PIM transformations. Transformations for refactoring (intra-formalism) or migration (inter-formalism) belong to this category.

- **Vertical**, which translate the model into another one at a different level of abstraction. Code generation can be seen as a translation from a higher (a model) to a lower (code) abstraction level. In this case, the target formalism is the language for which code is generated. In the context of MDA, PIM-to-PSM transformations are a kind of vertical transformations. Sometimes, the translation results in a partial model that the user should complete interactively. Other times, the generated code is combined with pre-existing libraries to form the complete application. This approach is useful for product family generation [16]. In the opposite direction, *reverse engineering* is a vertical transformation from lower (code) to higher (design) abstraction levels.

In order to express a model transformation, one has several options. We can look at three orthogonal characteristics in the transformation language: visual or textual, imperative or declarative, formal or semi-formal. One may find also hybrid approaches for each characteristics (i.e. languages which are declarative, but which also have imperative constructions). Our approach, graph transformation, lies in the category of visual, declarative and formal languages. One can find other combinations

to express the transformations. For example, using Java belongs to the category of textual, imperative and semi-formal languages.

Some properties are desirable for model transformation. For example *termination* in a finite amount of time, *confluence* (that is, from a source model a unique target model is generated, thus there is no non-determinism), *syntactic consistency* (the target model conforms to the given meta-model) and *semantic consistency* (some semantic properties – like behaviour – are preserved by the transformation). A formal approach to transformation allows the investigation of some of these properties. The basics of graph grammars and some of the variants that are useful for each kind of transformation are presented next, together with analysis techniques to analyze the abovementioned properties.

3 Graph Transformation

As models can be represented as attributed, typed graphs, graph transformation [4] is a natural means for their manipulation. Graph grammars are made of a set of rules and an initial graph. Rules are made of left and right hand sides (LHS and RHS), each one of them containing graphs. When applying a rule to a graph (called *host graph*), a match morphism must be found between the LHS and the host graph. If such a morphism is found, then the elements that are not preserved by the rule ($LHS - (LHS \cap RHS)$) are deleted in the graph, and the new elements ($RHS - (LHS \cap RHS)$) are added. The semantics of a graph grammar are all the possible reachable graphs that can be obtained applying the rules to the initial graph.

Figure 1 shows a graph transformation rule and its application to a model. The example is taken from the component-based modelling and simulation area, where models are made of interconnected components, that interact via timestamped events. The rule implements a small part of a simulator for the formalism. It checks if a composite component is receiving an event. In this case, it passes the event to an inner component connected to the input pin of the outer component. Below, the rule is applied to model G, yielding model H. Morphisms m and m^* are shaded in graphs G and H. In the rule, nodes with the same label are preserved (they belong to $LHS \cap RHS$).

One of the most popular formalizations of graph transformation is based on category theory [4]. There are two main approaches, the Single Pushout (SPO) and the Double Pushout (DPO) [3]. The latter is the one we follow in this paper. In the DPO approach, a rule is modelled by three components: $L \xleftarrow{l} K \xrightarrow{r} R$, where K are the preserved elements by the rule application. Without loss of generality, morphisms l and r can be injective, and thus $K = L \cap R$. A rule application can be modelled through two pushouts (a categorical construction, which for graphs is the union of two graphs through a common subgraph). The first one eliminates the elements in $L - K$, the second one adds the elements in $R - K$, as Figure 2 shows.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & & {\scriptstyle d}\downarrow & & \downarrow{\scriptstyle m^*} \\ G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H \end{array}$$

Figure 2: Direct Derivation as DPO construction.

Rules can be equipped with a set of *application conditions*, restricting the context in which they can be applied. A condition has a premise graph X, a set of consequent graphs Y_i, and morphisms y_i from X to each Y_i: $c = \{X, X \xrightarrow{y_i} Y_i\}$. If a match is found for X in the host graph, then a match has to be found for every consequent node Y_i for the rule to be applied. If a condition does not have consequent graphs, finding a match for the premise forbids the rule application. This is a special case of condition called *negative application condition* (NAC). On the contrary, if the condition has an empty premise, it is a positive application condition.

The execution of a graph grammar is nondeterministic: rules are tried at random in the host graph. There are two sources of nondeterminism: several rules may be applicable

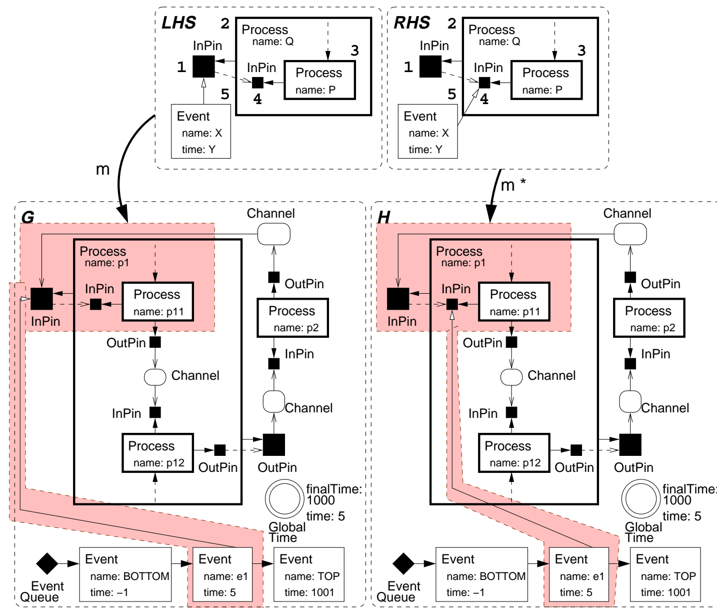

Figure 1: A Graph Transformation Rule.

at a certain moment, and several matches can be found for a single rule. In order to reduce the non-determinism, control structures for rule execution have been proposed. They range from assigning priorities to rules (the approach used in AToM^3 [11]), to using visual or textual connectives, similar to the ones for imperative programming languages.

There have been some attempts to increase rule expressivity. The first one combines graph transformation with meta-modelling [1]. The main idea is that when an object is present in a rule's LHS, then the object can be matched with any instance of the children classes of its classifier. In this way, a single rule is equivalent to a number of rules, substituting all the objects by instances of the classes in their inheritance clan.

Other graph transformation variant is called *parallel graph transformation* [19]. It allows specifying more complex patterns in both LHS and RHS, as certain parts of the rule can be replicated, synchronized by common parts. As defined in [19], a parallel rule is made of an *interaction scheme*, which consists of a set of *elementary rules*, a set of *subrules* and a set of the *embeddings* of the subrules into the elementary rules (that is, inclusions between their L, K and R parts). Thus an interaction scheme forms a bipartite graph. In this way, an elementary rule may be instantiated a number of times in the host graph, but all these matchings should overlap in the matching of all their subrules. Note how, each subrule or elementary rule can be equipped with arbitrary application conditions.

Figure 3 shows an example of a parallel rule that models sending events through the channels connected to the output pins. The subrule is called *Root Out Pin*, and is instantiated once in the host graph G. Then, elementary rule *All Out Channels* is instantiated twice, as there are two channels. Altogether, the parallel rule copies each event to each channel connected to the output port.

Several regular grammar rules would have been needed to model the same actions of the parallel rule in an iterative way. The par-

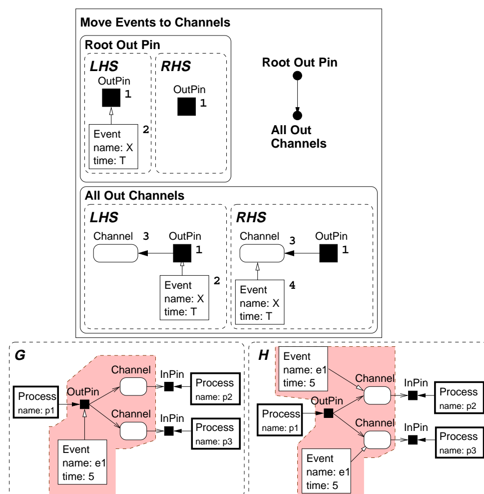

Figure 3: A Parallel Graph Grammar Rule.

allel rule is more efficient, and expresses the true concurrency of sending an event through a number of channels.

The categorical formalization of graph transformation has the advantage that the concrete category can be changed, and the results are still valid. Up to now, we have concentrated on rewriting regular graphs, but the same theoretical basis can be applied to other categories, such as graph triples. These are made of two graphs (G_i) and an intermediate graph ($LINK$) relating objects of both graphs. That is, nodes in $LINK$ have pairs of morphisms: one to G_1 objects and another one to G_2 objects. Thus a graph triple is depicted as $G_1 \xleftarrow{v_1} LINK \xrightarrow{v_2} G_2$. Graph triples can be manipulated by means of triple graph grammars [17] (TGGs). In [8] we defined TGGs using the DPO approach and used them in combination with meta-modelling to describe concrete-to-abstract syntax transformations, and consistency checkings for languages with multiple views. TGGs are also useful to specify inter-formalism model transformations, where the starting model is included in one graph, the target model in the other, while the correspondence graph relates concepts in both formalisms.

Note how, DPO rules are invertible (up to attribute computations). In this way, if one specifies an inter-formalism model transformation using rules, inverting LHS and RHS one gets a translator from the target formalism to the source formalism.

All the concepts introduced so far have been unified in AToM^3, as the meta-model in Figure 4 shows. In this way, a graph grammar is made of simple as well as parallel rules, each having a priority. They can rewrite normal as well as triple graphs. The *amalg_scheme* attribute in *Interaction Scheme* class controls the way in which elementary rules and subrules can be instantiated. In this paper, for space limitations we do not explain further this concept. The interested reader can consult [19].

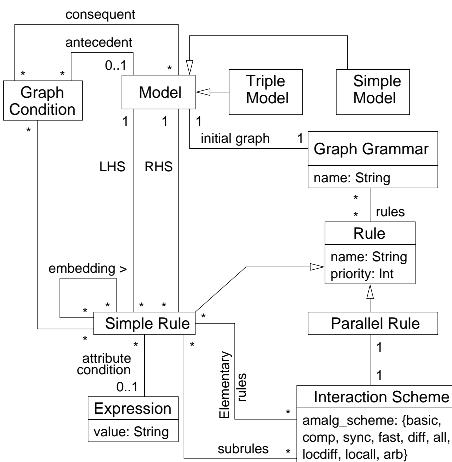

Figure 4: A Unifying Meta-model for Graph Transformation Variants.

3.1 Analysis Techniques

The categorical formalization of graph grammars allows analysing properties of the transformations themselves [12]. One of the properties that we are interested in is *termination* of the grammar execution, which is indecidible in general. Nonetheless, dividing the set of symbols and rules in layers, and restricting the symbol layer that a rule in a certain layer can

create and delete, it is possible to demonstrate termination [5]. In addition, confluent *transformations* [10] are those that, starting from an initial model, give as result a unique target model. That is, the result is deterministic (although one can have different paths in the grammar execution, all leading to the same result). Termination and confluence ensure a *functional behaviour* of the transformation. *Critical pair* analysis can be used to check confluence. Both, layers and critical pair analysis, are available in the AGG tool [20].

For inter-formalism transformations, we are interested in *syntactic* and *semantic consistency.* The former means that the target model should conform to the target formalism. The latter means that certain semantic properties of the source model should be preserved by the grammar execution. Both kind of analysis are discussed in [12].

Other interesting results for transformations implementing simulations are those concerning *concurrency* [2]. Concurrency can be studied from two points of view. In the *sequential approach*, a parallel computation can produce a number of interleavings. Here the concepts of *sequential* and *parallel* independence state when two rule applications can be performed in the reverse order. In the *explicit approach* to parallelism, actions are really simultaneous. The *parallelism theorem* states when two rules can be composed in a single *parallel rule* (thus no intermediate states are produced) and when a rule can be sequentialized.

Finally, when specifying a transformation, one can state *global constraints.* These model undesirable states that we do not want to happen in the system. These global constraints are expressed in form of graphs and can be translated into application conditions for each rule of the grammar [9]. In this way, the designer is sure that the system will not enter in a non-desirable state.

4 Example

As an example of the concepts introduced so far, we present a domain specific language for component-based modelling and simulation called **MiCo** (for Minimal Components). In this approach, models are made of components, each having an interface (input and output ports) and a behaviour. Components communicate by sending timestamped events through ports. In this way, the port type is given by the kind of events it can handle. Moreover, we are interested in multi-formalism modelling [21], where the behaviour of the different components may have to be described using different languages. In this way, the languages used for the description of the behaviour are left open, and can be extended, but up to now it can be done with *event graphs* [13].

We have defined five diagrams for this visual language. In the *Specification Diagram*, users can define components. These can be either simple components, or be composed of other components. In this diagram, one declares the ports, their types and the inner structure of the composite diagrams. Moreover, inheritance relationships can be established between components. In the *Connections Diagram*, users can specify constraints (regarding multiplicity) in the way the components can be connected through ports. This is an optional diagram, as one can just use the information of the port types. In the *Events Diagram* it is possible to define the event types and their attributes. In the *Behaviour Diagram* one can describe the behaviour of each simple component type. Note how the behaviour of a composite component is described in terms of its inner components. Finally, in the *Run Time Diagram*, it is possible to specify a run time configuration, made of component instances (that we call processes), which execute the behaviour defined in its component type.

Figure 5 shows some of the diagrams of the language, applied to the design of an interactive application. The application is made of cells, which can change their colour. Cells can be connected to each other and pass these "change colour" events. Cells may have different event passing behaviours (do not pass message and change colour, pass and do not change colour, etc.) The specification diagram contains a simple compo-

nent (*“Behaviour Controller”*), which is meant to be connected to a cell to control its behaviour. The event diagram shows the kind of events the system manages. Cells interchange *“Change”* events, while *“Delay”* events are sent to Cells by Delay controllers. The connections diagram specifies the kind of connections one may have, with the cardinalities. Ports that are connected in this diagram should be compatible in the kind of events they can send and receive. If we are not interested in restricting the default cardinality (minimum of zero and maximum of infinity), then this diagram is not necessary. In the run-time diagram, one can connect processes if the ports are compatible. If the connections diagram is present, then it provides extra constraints. The run-time diagram of the example shows a configuration with two connected cell processes and their corresponding controller processes.

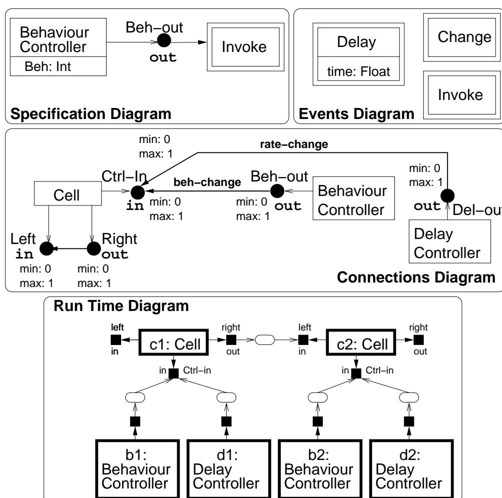

Figure 5: Some Diagrams of the MiCo Language.

Our objectives with this language were to make it as small as possible, as well as giving it a precise semantics. Thus, we avoided using “big” languages such as UML2.0 or SysML [18], as their complexity makes very hard to assign them a complete formal semantics. This is one of the advantages of “small” languages: they can be precisely described (sometimes by mapping them onto a semantic domain). Moreover, they contain powerful primitives for the specific domain and maximally constrain users for the task to be performed. In our case, using UML or SysML would lead to a big part of the language not being used.

We have defined the operational semantics of the language by means of graph transformation [13]. We have extended the previous work in [13] by defining views, consistency checkings between views, composite components and inheritance. Therefore we have had to extend the operational semantics for composite components and inheritance. Execution is based on a *future event queue*. The first event in the queue is executed by the corresponding component. The execution of the behaviour may produce sending events through ports. These are routed to the appropriate component and scheduled (ordered by time) in the event queue. Note how if an event reaches a composite component, the event is sent down in the hierarchy until it reaches a simple component (see rule in Figure 1).

5 Tool Support

AToM^3 [11] is a tool that allows defining visual languages by means of meta-modelling. In this way, starting from a meta-model, it generates a modelling environment for the specified VL. Figure 6 shows a snapshot of the tool containing a part of the meta-model for the definition of the MiCo language.

The tool also provides support to model languages with multiple views, that is, languages that contain several diagrams for describing different perspectives of a system. The views of the language are defined by selecting the necessary classes, associations, attributes and constraints from the complete meta-model. One modelling environment for all the views is generated. As the views can overlap, the tool assists in maintaining their syntactic consistency by creating a common and unique model (called *repository*) gluing all the views through the common elements. Note how this can be formalized in categorical terms as a pushout-star, which is a co-limit construc-

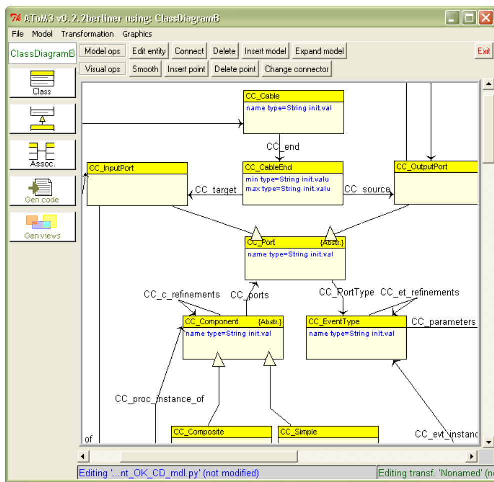

Figure 6: Meta-model of the MiCo Language.

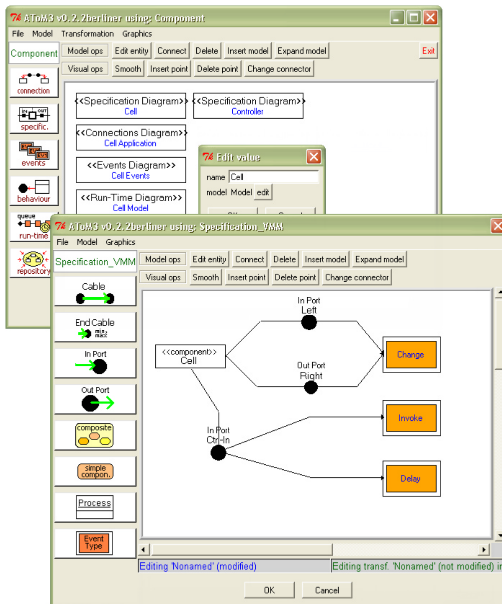

Figure 7: Generated Environment for MiCo.

tion. The *repository* is automatically built by the execution of triple graph grammars, which propagate changes from views to the repository, and vice versa if needed. The rules in these grammars relate the repository model and the different view models ($repository \xleftarrow{v_1} LINK \xrightarrow{v_2} view_i$). In the same way, users can incorporate their own specific consistency checkings between views also in the form of triple rules.

As an example, the upper window in figure 7 shows the generated environment for the MiCo language. Five views of the complete meta-model have been defined, corresponding to the five types of diagrams in the language. Users can add a new diagram of a certain type by clicking the corresponding button and then the canvas. Afterwards, the newly created diagram and also its attributes can be edited, as it is shown on the right of the same figure. Consistency among diagrams is achieved by the construction of the repository using the automatically generated triple graph grammars.

Model manipulation (simulation, optimization, translation into another formalisms and code generation) can be expressed in AToM^3 by means of graph rewriting. Both regular and triple graph grammars are supported. Grammars are defined as a list of rules. Each rule is composed by a LHS, a RHS, a set of application conditions (graphical or textual), and some actions to be performed after the rule application. Rules also have a priority, which controls the order in which rules are tried. Figure 8 shows the definition of a parallel rule for the simulation of MiCo models.

Some of the mechanisms explained in section 3 have been implemented in the tool in order to increase the expressivity of rules. In particular, the possibility of combining meta-modelling with graph transformation, as well as the use of parallel graph transformations, are available for both regular and triple graph grammar rules. The former can be activated by selecting *Subtypes Matching* in the rule attributes (the button box to the right of Figure 8). The latter can be activated by selecting *Amalgamate Rules* in the graph grammar attributes (the button box on the left in the

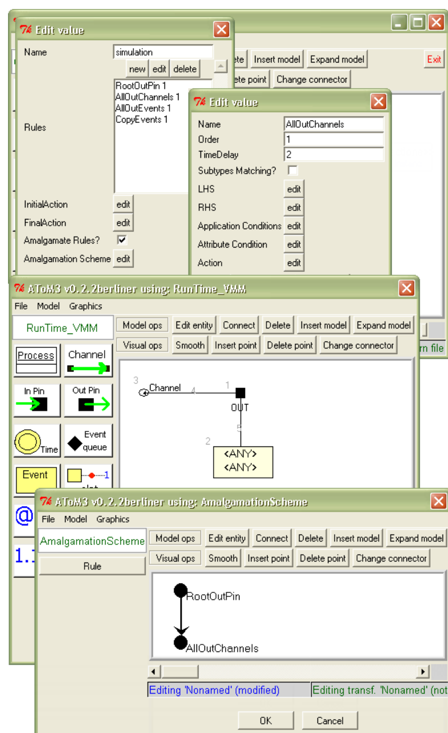

Figure 8: Defining a Parallel Rule in AToM^3

same figure). In order to define a parallel rule, the subrules and the elementary rules have to be defined separately. Figure 8 shows the definition of the parallel rule in Figure 3. It is made of elementary rule *All Out Channels* and subrule *Root Out Pin*. The mapping between subrules and elementary rules has to be specified, as the lower window in Figure 8 shows.

6 Related Work

There are many other meta-modelling tools able to describe and generate customized environments for visual languages, for example GME [14] or MetaEdit+ [16]. The novelty of our approach is that in AToM^3, model transformation can be expressed with a variety of formal graph transformation approaches, and that multiple views are supported.

Other approaches for model manipulation can be found in the proposals for OMG's MOF2.0 Query/Views/Transformation (see an overview of such submissions in [6]). Graph transformation can be useful for the transformations part. In practice, graph transformation is not purely declarative, as one usually adds a language for rule execution control on top of it. This necessity is more acute as transformations become more complex. In addition, for certain kind of transformations, such as some cases of code generation, a purely imperative textual approach is better suited. A neutral action language would be needed in this case. By now, in AToM^3 it is possible to use Python code.

7 Conclusions

In this paper, we have given an overview of current graph transformation techniques, and how they can be applied in the context of MDD. Some of these techniques are implemented in the AToM^3 tool. As an example, we have presented MiCo, a tiny component-based framework for modelling and simulation. The language has several views for structure and behaviour specification. It has a formal semantics based on graph transformation.

We are working in analysis techniques for graph transformation, and in mechanisms to provide consistency for dynamic semantics to multiple views environments. With respect to MiCo, we want to define extra views to describe the graphical appearance of the application. We are currently working in updating a code generator [13] for an older version of MiCo. In this way, it will be able to generate code for the complete application, once its behaviour has been analyzed.

References

[1] Bardohl, R., Ehrig, H., de Lara J., and Taentzer, G. 2004. *Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation*. LNCS 2984, pp.: 214-228. Springer.

[2] Baldan, P., Corradini, A., Ehrig, H., Löwe, M., Montanari, U. and Rossi, F., 1999. *Concurrent Semantics of Algebraic Graph Transformations.* Handbook of Graph Grammars and Computing by Graph Transformation (3), pp.: 107-187.

[3] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M. 1999. *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach.* In [4], pp.: 163-246

[4] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol(1)* World Scientific.

[5] Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S. 2005. *Termination Criteria for Model Transformation.* LNCS 3442, pp.: 49-63. Springer.

[6] Gardner, T., Griffin, C., Koehler, J., Hauser, R. 2003. *A review of OMG MOF2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard.* MetaModelling for MDA Workshop, York, England.

[7] Grunske, L., Geiger, L., Zündorf, A., Van Eetvelde, N., Van Gorp, P., Varró, D. 2005. *Using Graph Transformation for Practical Model Driven Software Engineering.* In Vol.II of Research and Practice in Software Engineering, Springer.

[8] Guerra, E., de Lara, J. 2004. *Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation.* LNCS 3256, pp.: 54-69. Springer

[9] Heckel, R., Wagner, A. 1995. *Ensuring consistency of conditional graph rewriting - a constructive approach.* Proc. of SEGRAGRA 1995, in ENTCS Vol 2, 1995.

[10] Heckel, R., Küster, J. M., Taentzer, G. 2002. *Confluence of Typed Attributed Graph Transformation Systems.* LNCS 2505, pp.: 161-176. Springer.

[11] de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* LNCS 2306, pp.: 174 - 188. Springer. See: `http://atom3.cs.mcgill.ca`

[12] de Lara, J., Taentzer, G. 2004. *Automated Model Transformation and its Validation with AToM^3 and AGG.* LNAI 2980. Springer, pp.: 182-198.

[13] de Lara, J. 2004. *Distributed Event Graphs: Formalizing Component-based Modelling and Simulation.* Visual Languages and Formal Methods, in ENTCS (Elsevier), Vol 127(4), pp.: 145-162.

[14] Lédczi, A., Bakay, A., Marói, M., Vögyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G. *Composing Domain-Specific Design Environments.* IEEE Computer, Nov. 2001, pp.: 44-51.

[15] Mens, T., Czarnecki, K., Van Gorp, P. 2004. *A Taxonomy of Model Transformations.* Proc. Dagstuhl Seminar 04101.

[16] Pohjonen, R., Tolvanen, J-P. 2002. *Automated Production of Family Members: Lessons Learned* OOPSLA workshop on Product Line Engineering.

[17] Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars.* LNCS 903, pp.: 151-163. Springer.

[18] SysML web at `http://www.sysml.org/`

[19] Taentzer, G. 1996. *Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems.* PhD Thesis, Shaker Verlag.

[20] Taentzer G., Ermel C., Rudolf M. 1999 *The AGG Approach: Language and Tool Environment*, In Handbook of Graph Grammars and Computing by Graph Transformation (2).

[21] Vangheluwe, H., de Lara, J., Mosterman, P. 2002. *An Introduction to Multi-Paradigm Modelling and Simulation.* In AI, Simulation and Planning, pp.: 9-20.