Computer Aided Multi-Paradigm Modelling to Process Petri-Nets and Statecharts

Juan de Lara[1] and Hans Vangheluwe[2]

[1] ETS Informática
Universidad Autónoma de Madrid
Madrid Spain,
Juan.Lara@ii.uam.es
[2] School of Computer Science
McGill University, Montréal
Québec, Canada
hv@cs.mcgill.ca

Abstract. This paper proposes a *Multi-Paradigm* approach to the modelling of complex systems. The approach consists of the combination of meta-modelling, multi-formalism modelling, and modelling at multiple levels of abstraction. We implement these concepts in AToM^3, A Tool for Multi-formalism, Meta-Modelling. In AToM^3, modelling formalisms are modelled in their own right at a meta-level within an appropriate formalism. AToM^3 uses the information found in the meta-models to automatically *generate tools* to process (create, edit, check, optimize, transform and generate simulators for) the models in the described formalism. Model processing is described at a meta-level by means of models in the graph grammar formalism. As an example, meta-models for both syntax and semantics of Statcharts (without hierarchy) and Petri-Nets are presented. This includes a graph grammar modelling the transformation between Statecharts and Petri-Nets.

Keywords: Modelling & Simulation, Meta-Modelling, Multi-Formalism Modelling, Automatic Code Generation, Graph Grammars.

1 Introduction

Complex systems are characterized, not only by a large number of components, but also by the diversity of these components. This often implies the components are described in different formalisms. Several approaches are possible to deal with this variety:

1. A single *super-formalism* may be constructed which subsumes all the formalisms needed in the system description. In most cases, this is neither possible nor meaningful.
2. Each system component may be modelled using the most appropriate formalism and tool. In the *co-simulation* approach, each component is subsequently

simulated with a formalism-specific simulator. Interaction due to component coupling is resolved at the trajectory level. Questions about the overall system can only be answered at the state trajectory level. It is no longer possible to answer symbolic, high-level questions which could be answered within the individual components' formalisms.

3. In *multi-formalism* modelling, as in co-simulation, each system component may be modelled using the most appropriate formalism and tool. However, a single formalism is identified into which each of the component models may be symbolically transformed [19]. The formalism to transform to depends on the question to be answered about the system. The Formalism Transformation Graph (FTG, see Figure 1) proposed by the authors suggests DEVS [21] as a universal common formalism for simulation purposes. It is easily seen how multi-formalism modelling subsumes both the super-formalism approach and the co-simulation approach.

In order to make the multi-formalism approach applicable, we still have to solve the problem of interconnecting a plethora of different tools, each designed for a particular formalism. Also, it is desirable to have highly problem-specific formalisms and tools. The time needed to develop these is usually prohibitive. We tackle this problem by means of *meta-modelling*. Using a meta-layer of modelling, it is possible to model the modelling formalisms themselves. Using the information in these meta-layers, it is possible to generate customized tools for models in the described formalisms. The effort required to construct a tool for modelling in a formalism tailored to particular applications thus becomes minimal. Furthermore, when the generated tools use a common data structure to internally represent the models, transformation between formalisms is reduced to the transformation of these data structures.

In this article, we present AToM^3 [2], a tool which implements the ideas presented above. AToM^3 has a meta-modelling layer in which different formalisms are modelled. From the meta-specification (in the Entity Relationship formalism extended with constraints), AToM^3 generates a tool to process models described in the specified formalism. Models are represented internally using *Abstract Syntax Graphs*. As a consequence, transformations between formalisms is reduced to graph rewriting. Thus, the transformations themselves can be expressed as graph grammar models [5]. Although graph grammars have been used in highly diverse areas such as graphical editors, code optimization, and computer architecture. [7], to our knowledge, they have never been applied to formalism transformations. In this paper, we present an example of transforming Statechart models (without hierarchy) into behaviourally equivalent Petri-Nets.

The rest of the paper is organized as follows: section 2 introduces the concepts of Multi-Paradigm Modelling and section 3 presents other related approaches. Section 4 gives an overview of the multi-paradigm modelling tool AToM^3. In section 5, we show an example of model manipulation in AToM^3. Finally, section 6 presents conclusions and future work.

2 Multi-Paradigm Modelling

Computer Automated Multi-Paradigm Modelling is an emerging field which addresses and integrates three orthogonal directions of research:

1. *Multi-Formalism modelling*, concerned with the coupling of and transformation between models described in different formalisms. In Figure 1, a part of the "formalism space" is depicted in the form of an FTG. The different formalisms are shown as nodes in the graph. The arrows denote a homomorphic relationship "can be mapped onto". The mapping consists of transforming a model in the source formalism into a behaviourally equivalent one in the target formalism.

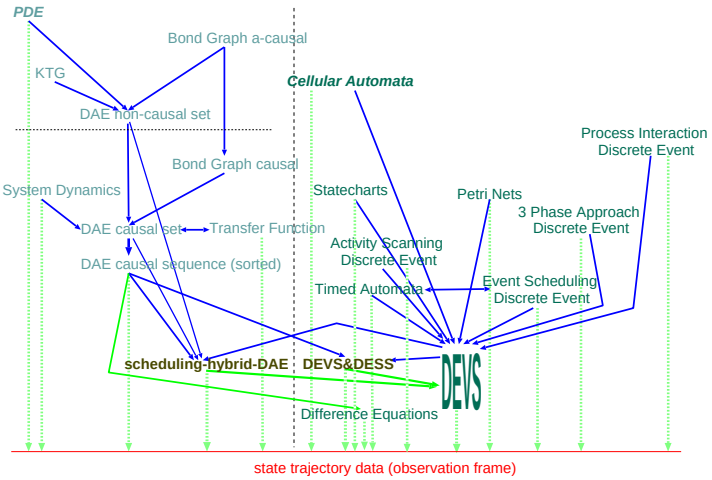

Fig. 1. Formalism Transformation Graph (FTG).

In our approach, we allow the specification of composite systems by coupling heterogeneous expressed in different formalisms. For the analysis of its properties, most notably, its behaviour, the composite system must be assessed by looking at the *whole* multi-formalism system. For appropriate processing (simulation, code generation, etc.) of the composite model, its components may have to be transformed to a common formalism, which can be found in the FTG. As we will see later, formalisms are meta-modelled and stored as graphs. Thus, the transformations denoted by the arrows of the FTG can be modelled as graph grammars.

2. *Model Abstraction*, concerned with the relationship between models at different levels of abstraction.
3. *Meta-Modelling* (models of models), which is the process of modelling formalisms. Formalisms are described as models using meta-formalisms. The

latter are nothing but formalisms expressive enough to describe other formalisms' syntax and semantics. Examples are the Entity Relationship formalism or UML class diagrams. A model of a meta-formalism is called a meta-meta-model; a model of a formalism is called a meta-model. Table 1 depicts the levels considered in our meta-modelling approach.

Level	**Description**	**Example**
Meta-Meta-Model	Model that describes a formalism that will be used to describe other formalisms.	Description of Entity-Relationship Diagrams, UML class Diagrams
Meta-Model	Model that describes a simulation formalism. Specified under the rules of a certain Meta-Meta-Model	Description of Deterministic Finite Automata, Ordinary differential equations (ODE)
Model	Description of an object. Specified under the rules of a certain Meta-Model	$f'(x) = -\sin x, f(0) = 0$ (in the ODE formalism)

Table 1. Meta-Modelling Levels.

To be able to fully specify modelling formalisms, the meta-level formalism may have to be extended with the ability to express constraints (limiting the number of meaningful models). For example, when modelling a Determinsitic Finite Automaton, different transitions leaving a given state must have different labels. This cannot be expressed within Entity-Relationship diagrams alone. Expressing constraints is most elegantly done by adding a constraint language to the meta-modelling formalism. Whereas the meta-modelling formalism frequently uses a graphical notation, constraints are concisely expressed in textual form. For this purpose, some systems [18] (including ours) use the Object Constraint Language OCL [15] used in the UML. As AToM^3 is implemented in the scripting language Python [17], arbitrary Python code may also be used.

3 Other approaches

A similar approach to our vision of Multi-Paradigm Modelling (although oriented to the software engineering domain) is ViewPoint Oriented Software Development [9]. Some of the concepts introduced by ViewPoint Oriented Software Development have a clear counterpart in our approach (for example, *ViewPoint templates* are equivalent to meta-models). They also introduce the relationships between ViewPoints, which are similar to our coupling of models and graph transformations.

Other approaches to interconnecting formalisms are Category Theory [8], in which formalisms are cast as categories and their relationships as functors. See also [20] and [14] for other approaches.

There are other visual tools to describe formalisms using meta-modelling, among them DOME [4], Multigraph [18], MetaEdit+ [12] and KOGGE [6]. Some of these allow one to express formalism semantics by means of a textual language (KOGGE for example uses a language similar to Modula-2). Our approach is quite different, as we express such semantics by means of graph grammar models. We believe that graph grammars are a natural, declarative, and general way to express transformations. As graph grammars are highly amenable to graphical representation, they are superior to a purely textual language. Also, none of the tools consider the possibility of "translating" models between different formalisms.

There are various languages and systems for graph grammar manipulation, such as PROGRES [16], GRACE [10] and AGG [1]. None of these have a meta-modelling layer.

Our approach is original in the sense that we combine the advantages of meta-modelling (to avoid explicit programming of customized tools) and graph transformation systems (to express tool behaviour and formalism transformation). Our main contribution is in the field of multi-paradigm modelling [19], as we have a general means to transform models between different formalisms.

4 AToM^3: an overview

AToM^3 is a tool written in Python [17] which uses and implements the concepts presented above. Its architecture is shown in Figures 2 and 3. In both figures, models are represented as white boxes, having in their upper-right hand corner an indication of the meta-...model they were specified with. It is noted that in the case of a graph grammar model, to convert a model in formalism F_{source} to a model in formalism F_{dest}, it is necessary to use the meta-models of both F_{source} and F_{dest} together with the meta-model of graph grammars.

The main component of AToM^3 is the *Kernel*, which is responsible for loading, saving, creating and manipulating models (at any meta-level, via the *Graph-Rewriting Processor*), as well as for generating code for customized tools. Both meta-models and meta-meta-models can be loaded into AToM^3 as shown in Figure 2. The first kind of models allows constructing valid models in a certain formalism, the second kind are used to describe the formalisms themselves. Models, meta-models and meta-meta-models are all stored as *Abstract Syntax Graphs* whose nodes and links are typed, and their relationships are subject to constraints dictated by the formalism under which these models were defined.

The ER formalism extended with constraints is available at the meta-meta-level. It is perfectly possible to define other meta-meta-formalisms using ER, such as UML class diagrams. Constraints can be specified as OCL or Python expressions, and the designer must specify when (pre- or post- and on which event) the condition must be evaluated. Events can be *semantic* (such as editing

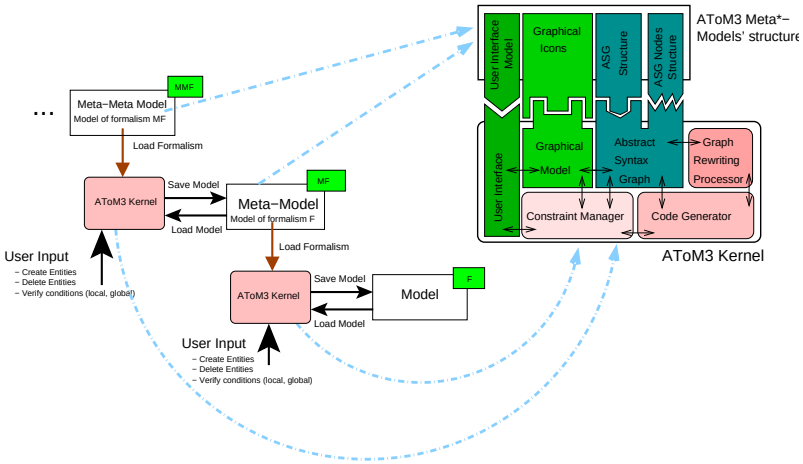

Fig. 2. Meta-... Modelling in AToM^3.

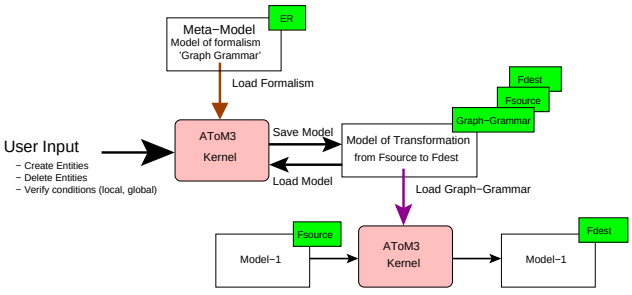

Fig. 3. Model Transformation in AToM^3.

an attribute, or connecting two entities) or *graphical* (such as dragging, dropping, or moving an object).

When modelling at the meta-meta-level, the entities which may appear in a model must be specified together with their attributes. We will refer to this as the *semantic* information. For example, to define the Petri-Nets Formalism, it is necessary to define both *Places* and *Transitions*. Furthermore, for *Places* we need to add the attributes *name* and *number of tokens*. For *Transitions*, we need to specify their *name* attribute.

In the meta-model, it is also possible to specify the *graphical appearance* of each entity when instantiated at the lower meta-level. For example, for Petri-Nets, we can choose to represent *Places* as circles with the *number of tokens* inside the circle and the *name* beside it, and *Transitions* as thin rectangles with the *name* beside them. That is, we can specify how some semantic attributes are displayed graphically. Constraints can also be associated with the graphical entities.

The meta-meta-information is used by the *Kernel* to generate some Python files (see the upper-right corner of Figure 2), which, when loaded by the *Kernel*, allows the processing of models in the defined formalism. These files include a model of the user interface presented when the formalism is loaded. This model follows the rules of the *"Buttons"* formalism, and by default contains a "create" button for each object found in the meta-model. For the case of the Petri-Net formalism, it contains buttons to create *Places*, *Transitions*, and the connections between them. This model can be modified using AToM^3 to for example add buttons to execute graph grammars on the current model or to delete unwanted buttons. When a formalism is loaded, the *Kernel* interprets the user interface model, to create and place the actual widgets and associate them with the appropriate actions.

Figure 4 shows an example of meta-modelling at work. It shows AToM^3 being used to describe the Petri-Net formalism (left), and the automatically generated tool (from the previous description) to process Petri-Nets.

For the implementation of the Graph Rewriting Processor, we have used an improvement of the algorithm given in [5], in which we allow non-connected graphs in Left Hand Sides (LHS) in rules. It is also possible to define a sequence of graph grammars to be applied to the model. This is useful, for example to couple grammars to convert a model into another formalism, and then apply an optimizing grammar. Often, for clarity and efficiency reasons, graph grammars are divided into different independent parts.

Rule execution can either be continuous (no user interaction) or step-by-step whereby the user is prompted after each rule execution. As the LHS of a rule can match different subgraphs of the host graph, we can also control whether the rule must be applied to all the matching subgraphs (if disjoint), if the user can choose one of the matching subgraphs interactively, or whether the system chooses one at random.

As in grammars for formalism transformations we have a mixing of entities belonging to different formalisms, it must be possible to open several meta-

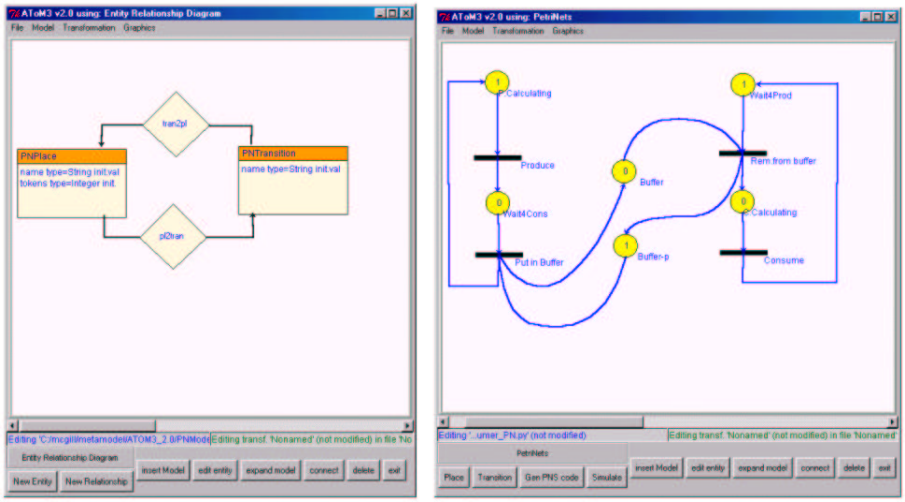

Fig. 4. Meta-Model of the Petri-Net Formalism (left) and Generated Tool to Process Petri-Net Models, with a Producer-Consumer model (right).

models at the same time (see Figure 3). Obviously, the constraints of the individual formalism meta-models are meaningless when entities in different formalisms are present in a single model. Such a model may come to exist during the intermediate stages of graph grammar evaluation when transforming a model from one formalism into another. It is thus necessary to disable evaluation of constraints during graph grammar processing (i.e., all models are reduced to Abstract Syntax Graphs). At the end of the execution of a graph grammar for formalism transformation, the Kernel checks if the resulting model is valid in some of the currently open formalisms, and closes the other formalisms.

5 Model manipulations

In this section we present some examples of the kind of model manipulations performed in AToM^3. These manipulations complement the default Kernel capabilities. When a tool is generated, it can be used to build, load and save models and check whether they are correct. Graph grammars are a way to enrich these functionalities. Some examples of the uses of graph grammars in AToM^3 are:

- Formalism transformation: n example will be presented in subsection 5.1.
- Model optimization: these transformations reduce the complexity of a model.
- Simulator specification: allow us to express the operational semantics of a formalism.
- Code generation: permit the generation of code for a specific tool.

5.1 Formalism Transformation

A formalism transformation takes a model m_1 in a formalism F_{source} and converts it into a model m_1' but expressed in formalism F_{dest}. The FTG (see Figure 1) shows behaviour preserving transformations between formalisms. We have implemented several of these transformations in AToM^3 with graph grammars. Some of the reasons to perform formalism transformations are:

- In a composite model with components described in different formalisms, it may be possible to transform each component into some formalism reachable (in the FTG) from the formalisms of each component. Then, once the composite model has all its components described in the same formalism, we can simulate it.
- To solve problems that are easier to solve in some other formalism. For example, in the case of a model in the Statecharts formalism, for which we want to determine whether it can deadlock. As this kind of analysis is well known for Petri-Nets, we can transform the Statecharts model into the Petri-Nets formalism, and then solve the problem in that domain. We will provide such a transformation (limited to non-hierarchical Statecharts) in this section.

The Statecharts formalism is widely used for modelling reactive systems. Statecharts can be described as an elaboration of the HiGraphs [11] semantics as well as of State Automata. HiGraphs allow for hierarchy (blobs can be inside blobs), parallelism (a blob can be divided into concurrent Orthogonal components), Blobs are connected via hyperedges. Our meta-model for Statecharts is thus composed of the following entities:

- *Blobs*. These have a *name* and represent the states in which the system can be. A state may have one or more *Orthogonal components* inside. We represent *Blobs* as rectangles.
- *Orthogonal components*. These have a *name* and a Statechart inside. We will represent *Orthogonal components* as rectangles with rounded, dashed lines. As we will see later, *"insideness"* is represented as a relationship at the meta-level.
- *Initial state*. These kind of entities mark the state in which the system enters when reaching a certain *Orthogonal component*.

The following relationships are also included in the meta-model:

- *Hyperedge*. This relationship implements a directed hyperedge and is used to connect a group of *Blobs*. It contains the following attributes:
  - *Event*, which is a list of the events that must occur for the transition to take place;
  - *Broadcast_Event*, which is a list of the events to broadcast if the transition takes place;
  - and *Actions* which stores Python code to be executed when the transition takes place.

Events can be global, or can be directed to a particular *Orthogonal component*. Also included in the *Event* list is a guard: the condition of a particular *Orthogonal component* being in a certain state.

- *has_Inside*. This is a relationship between *Orthogonal components* and *Blobs*. It expresses the notion of hierarchy: *Blobs* are inside *Orthogonal components*.
- *composed_of*. This is a relationship between *Blobs* and *Orthogonal components*. It expresses the notion of hierarchy in the other direction, meaning that *Blobs* are composed of one or more *Orthogonal components*.
- *has_Initial*. This relationship express the notion of hierarchy between *Orthogonal components* and *Initial states*.
- *iconnection*. This relationship allows the connection of an *Initial state* and a *Blob*

As relationships *has_Inside*, *composed_of* and *has_Initial* are a means to express hierarchy they are drawn as invisible links.

Some actions have been added to the meta-model to move all the *Blobs* inside of *Orthogonal components* when the latter are moved. Also, when an *Orthogonal component* is placed inside a *Blob*, this is enlarged to accomodate it. Figure 5 shows the meta-model and the generated tool for modelling Statecharts.

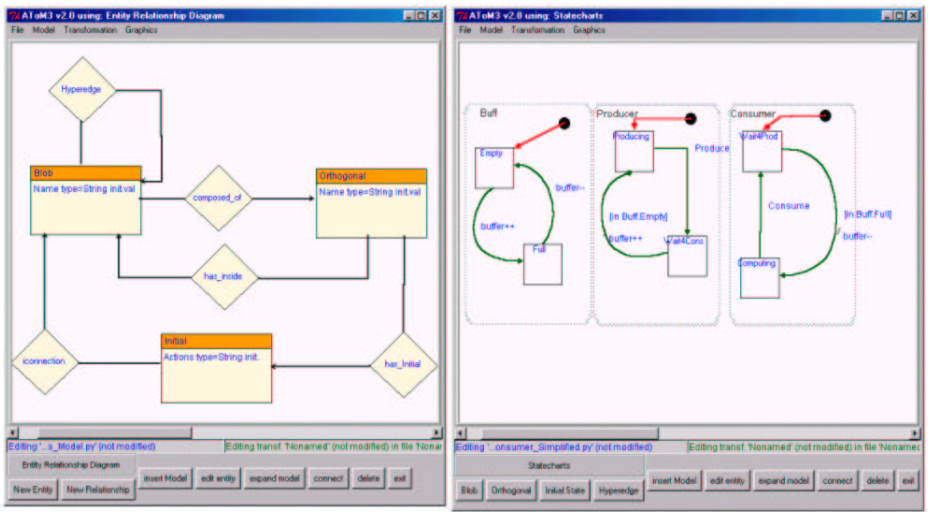

Fig. 5. The Meta-Model (left) and the generated tool for Statechart modelling. Used here to model the Producer-Consumer Problem (right).

For brevity, in this section we will present a graph grammar to transform non-hierarchical Statecharts into Petri-Nets. That is, a Statechart model can be composed of several *Orthogonal components*, which can contain *Blobs*, but the converse is not allowed. The model in Figure 5 has this restriction.

This transformation has been implemented in four different, independent graph grammars. This, as there are four well defined steps for completing the transformation, and there is no need to evaluate the applicability of the other rules, as none of them will be applicable. This makes the transformation process faster. Another reason to split the transformation is that it is easier to debug. The four graph grammars are:

1. The first graph grammar identifies the events in the Statechart by looking at the events each *hyperedge* has in its lists. It creates a *Place* for each event. In our approach, each type of event in the system is assigned a *Place*. These are considered *interfaces* in which the system puts a token whenever the event takes place. For each *Orthogonal component* in the Statechart model, the graph grammar creates *"local"* versions of the global events. Each global event is connected to the local events via a unique *Transition*, in such a way that when the global event is present, the token is broadcast to all the local events of the *Orthogonal components*. This global/local distinction is useful when an output event has to be sent:
   - If the event has to be sent to a particular *Orthogonal component*, a token will be placed in its corresponding local event *Place*.
   - If it has to be broadcast to the whole system, it will be placed in the global event *Place*.

   Note also that the local event *Places* are connected to a *Transition* whose purpose is to eliminate the token if it has not been consumed immediatelly after the event took place. This graph grammar is shown in Figure 6.

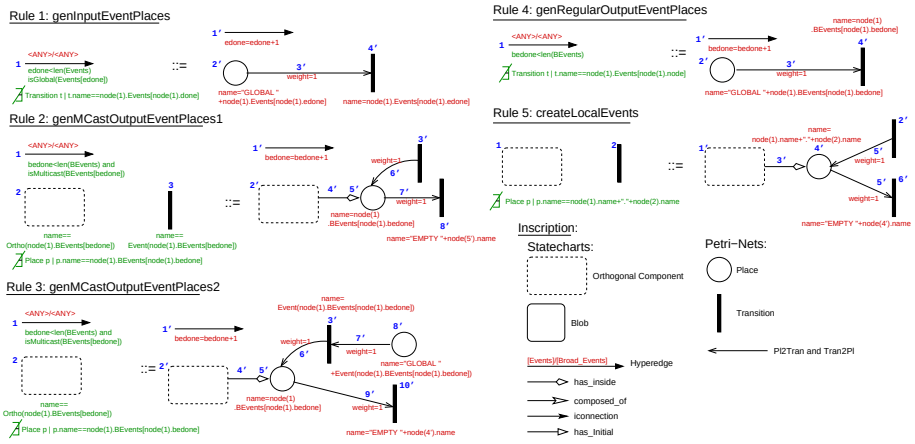

Fig. 6. First Graph Grammar: creates the Local and Global Event Places.

It can also be observed how the local events are connected to their corresponding *Orthogonal component*.

2. The second graph grammar creates a *Place* for each *Blob* and moves the *Hyperedges* from the *Blobs* to the *Places*. Figure 7 shows this graph grammar. The way the transfer of the *hyperedges* from the *Blobs* to the *Places*

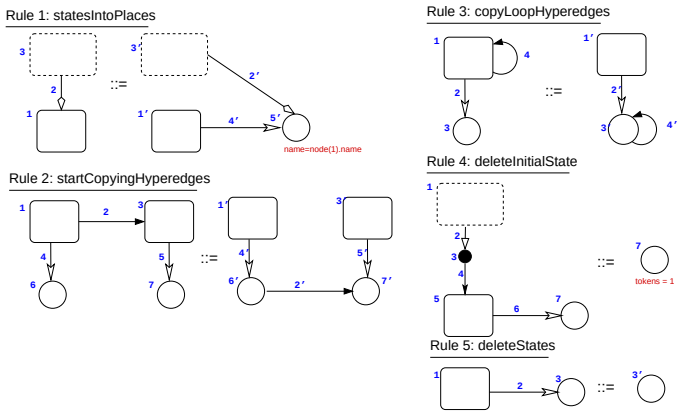

Fig. 7. Second Graph Grammar: creates the Local and Global Event Places.

is done is by keeping a connection from the *Blob* to the associated *Place*. The graph grammar also locates the initial state and thus puts a token in the corresponding *Place* (see rule 4). Once all the *hyperedges* are moved, the *Blobs* can be erased. During the execution of the Petri-Net, in the *Places* that represent the states of the system, we will have a number of tokens equal to the number of *Othogonal components*.

3. The third graph grammar converts the *hyperedges* into *Transitions* connected to the appropriate *Places*. Figure 8 shows some of the rules of this graph grammar (the implementation is composed of nine rules). The way to proceed is first to identify the *Place* associated with the first event of the list of events of the *hyperedge* and then make a connection between this event and an itermediate *Transition* which is created in the process, and then eliminate the event from the list. A similar process has to be followed for the lists of events to be broadcast. The process finishes when both lists of events are empty. Observe that we used a Prolog-like notation for identifying the first element of the lists of events and its subsequent elimination.
   Other rules of this graph grammar deal with the case when an *Orthogonal* component is in a certain state; and with the case of an event to be sent to a particular *Orthogonal component*.
4. The fourth graph grammar simply removes the *Orthogonal components*.

The results of the application of this graph grammar to the Statechart model in Figure 5 is shown in Figure 9 (after applying a graph grammar for simplifying Petri-Nets, not shown in this paper).

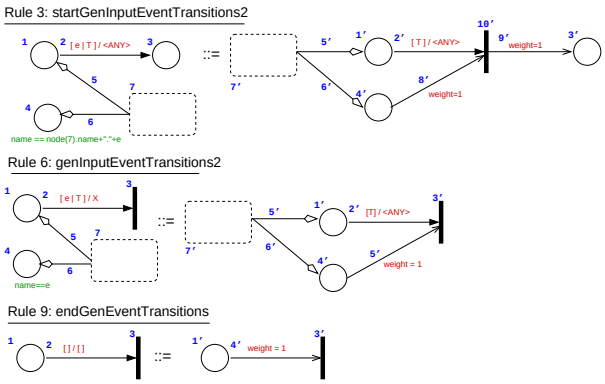

Fig. 8. Third Graph Grammar (not Completelly Shown): eliminates Hyperedges.

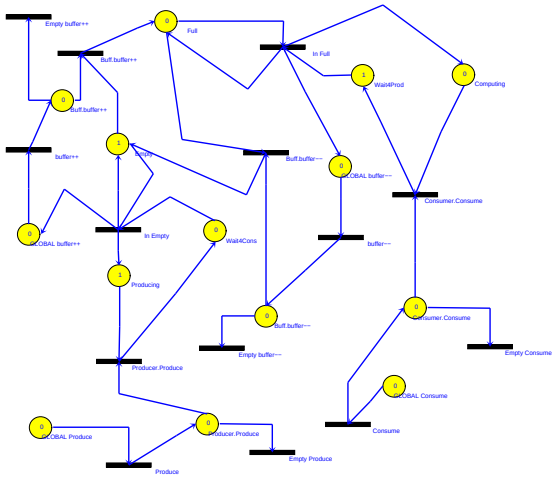

Fig. 9. Result of the application of the graph grammat to the Producer-Consumer Statechart.

There are other approaches to the conversion of Statecharts into Petri-Nets. In [13], a manual method is proposed, but it is not systematic. Basically, a human Petri-Net and Statecharts expert would have to understand the behaviour of the Statechart model and then model an equivalent Petri-Net. An equivalence proof between both models is not provided. The possibility of further simplification and manipulation of the model is not considered. Also, events are represented as *Transitions* whereas in our approach, we insert *interface Places* to represent *Global* and *Local* events. This facilitates the reuse of the Petri-Net models and makes it possible to send events to particular orthogonal components.

6 Conclusions and future work

In this paper we have discussed the advantages of a multi-paradigm approach when modelling complex systems. Meta-Modelling means explicitly model the formalisms. It allows for the automatic generation of customized tools. Formalism transformation permits the translation of models between formalisms to solve problems that are easier to solve in other formalisms.

We have presented AToM^3, a meta-modelling tool able to generate customized, formalism-specific tools. As models are stored in the form of graphs, AToM^3 can manipulate them using graph grammars. Users can define –in a visual, high level way– graph grammars to manipulate their models without having to modify or have knowledge about the AToM^3 kernel code.

In particular, we have shown how the user can analyze a Statechart model by converting it into a Petri-Net. Some other graph grammars could then be applied to reduce its complexity, and to simulate it or to generate code for a specific Petri-Net tool for further processing. Note how this process can be completely automated from the appropriate graph grammars. The process could also be made invisible to the Statecharts modeller, who could be only interested in knowing –in the case of the example presented– whether the Buffer can be made to exceed its capacity.

The advantages of using an automated tool for generating customized model-processing tools are clear: instead of building the whole application from scratch, it is only necessary to specify –in a graphical manner– the kind of models we will deal with. The processing of such models can be expressed by means of graph grammars, at the meta-level. Our approach is also highly applicable if we want to work with a slight variation of some formalism, where we only have to specify the meta-model for the new formalism and a tranformation into a "known" formalism (one that already has a simulator available, for example). We then obtain a tool to model in the new formalism, and are able to convert models in this formalism into the other for further processing. We have not only described formalisms commonly used in the simulation of dynamical systems, but we have also described formalisms such as Data Flow Diagrams and Structure Charts used for the structured description of software.

In the future, we plan to extend the tool in several ways:

– Describing another meta-meta-model in terms of the current one (the Entity-Relationship meta-meta-model) is possible. In particular, we plan to describe UML class diagrams. For this purpose, relationships between classes such as *inheritance* should be described. Thanks to our meta-modelling approach, we will be able to describe different subclassing semantics and their relationship with subtyping. Furthermore, as the semantics of inheritance will be described at the meta-level, code can be generated in non-object-oriented languages.
– Exploring the automatic proof of behavioural equivalence between two models in different formalisms by bi-simulation. This may help in validating that a graph grammar for formalism transformation is correct.
– Integrating a module to help the user to decide which alternatives are available a a certain moment of the modelling of a multi-formalism system. This module may assist in deciding to which formalism to transform each component (following the FTG).
– Extending the tool to allow collaborative modelling. This possibility as well as the need to exchange and re-use (meta-...) models raises the issue of formats for model exchange. A viable candidate format is XML.

References

1. AGG Home page: *http://tfs.cs.tu-berlin.de/agg/*
2. AToM^3 home page: http://moncs.cs.mcgill.ca/MSDL/research/projects/ATOM3.html
3. Blonstein, D., Fahmy, H., Grbavec, A.. 1996. *Issues in the Practical Use of Graph Rewriting*. LNCS 1073, Springer, pp.38-55.
4. DOME guide. http://www.htc.honeywell.com/dome/, Honeywell Technology Center. Honeywell, 1999, version 5.2.1
5. Dorr, H. 1995. *Efficient Graph Rewriting and its implementation*. LNCS 922, Springer.
6. Ebert, J., Sttenbach, R., Uhe, I. *Meta-CASE in Practice: a Case for KOGGE* Proceedings of the 9th International Conference, CAiSE'97, Barcelona. LNCS 1250, 203-216, Berlin, 1997. See KOGGE home page at: *http://www.uni-koblenz.de/∼ist/kogge.en.html*
7. Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) 1991. *Graph Grammars and their application to Computer Science: 4th International Workshop, Bremen, Germany, March 5-9, 1990*. LNCS 532, Springer.
8. Fiadeiro, J.L., Maibaum, T. 1995. *Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality* Proc.3rd Symposium on the Fundations of Software Engineering, G.E.Kaiser(ed),pp.: 72-80, ACM Press.
9. Finkelstein, A., Kramer, J., Goedickie, M. 1990. *ViewPoint Oriented Software Development* Proc, of the 3rd Workshop on Software Engineering and its Applications, Tolouse.
10. GRACE Home page: *http://www.informatik.uni-bremen.de/theorie/GRACEland/GRACEland.html*
11. Harel, D. On visual formalisms. *Comm. of the ACM*, 31(5):514–530, 1988.
12. Kelly, S., Lyytinen, K., Rossi, M. *MetaEdit+: A fully configurable Multi-User and Multi-Tool CASE and CAME Environment* In Advanced Information System Engineering; LNCS 1080. Berlin, Springer 1996. See MetaEdit+ Home page at: *http://www.MetaCase.com/*

13. King, P., Pooley, R. *Using UML to Derive Stochastic Petri Net Models* In Davies and Bradley Editors. UKPEW'99, Proc. 15th UK Performance Engineering Workshop. Bristol. pp.: 45-56.
14. Niskier, C., Maibaum, T., Schwabe, D. 1989 *A pluralistic Knowledge Based Approach to Software Specification* 2nd European Software Engineering Conference, LNCS 387, Springer, pp.:411-423.
15. OMG Home Page: *http://www.omg.org*
16. PROGRES home page: *http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html*
17. Python home page: *http://www.python.org*
18. Sztipanovits, J., et al. 1995. *"MULTIGRAPH: An architecture for model-integrated computing"*. In ICECCS'95, pp. 361-368, Ft. Lauderdale, Florida, Nov. 1995.
19. Vangheluwe, H. *DEVS as a common denominator for multi-formalism hybrid systems modelling.* In *IEEE International Symposium on Computer-Aided Control System Design*, pp.:129–134. IEEE Computer Society Press, September 2000. Anchorage, Alaska.
20. Zave, P., Jackson, M. 1993. *Conjunction as Composition* ACM Transactions on Software Engineering and Methodology 2(4), 1993, 371-411.
21. Zeigler, B., Praehofer, H. and Kim, T.G. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press, 2nd ed., 2000.