ATOM^3: A Tool for Multi-formalism Modelling and Meta-modelling

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

[1] ETS Informática
Universidad Autónoma de Madrid
Madrid Spain, fax: +34 91 348 22 35
Juan.Lara@ii.uam.es, jlara@cs.mcgill.ca
[2] School of Computer Science
McGill University, Montréal
Québec, Canada, fax: +1 (514) 398 38 83
hv@cs.mcgill.ca

Abstract. This article introduces the combined use of *multi-formalism* modelling and *meta-modelling* to facilitate computer assisted modelling of complex systems. The approach allows one to model different parts of a system using different formalisms. Models can be automatically converted between formalisms thanks to information found in a *Formalism Transformation Graph* (FTG), proposed by the authors. To aid in the automatic generation of multi-formalism modelling tools, formalisms are modelled in their own right (at a meta-level) within an appropriate formalism. The above approach has been implemented in the interactive tool ATOM^3. This tool is used to describe formalisms commonly used in the simulation of dynamical systems, as well as to generate custom tools to process (create, edit, transform, simulate, optimize, ...) models expressed in the corresponding formalism. ATOM^3 relies on graph rewriting techniques and graph grammars to perform the transformations between formalisms as well as for other tasks, such as code generation or simulator specification.

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

1 Introduction

Modelling complex systems is a difficult task, as such systems often have components and aspects whose structure as well as behaviour cannot be described in a single formalism. Examples of commonly used formalisms are Differential-Algebraic Equations (DAEs), Bond Graphs, Petri Nets, DEVS, Entity-Relationship diagrams (ERDs), and State Charts. Several approaches are possible:

- A single super-formalism may be constructed which subsumes all the formalisms needed in the system description. This is not possible nor meaningful in most cases, although there are some examples of formalisms which

span several domains (e.g. Bond Graphs for the mechanical, hydraulic and electrical domains.)

- Each system component may be modelled using the most appropriate formalism and tool. To invesitate the overall behaviour of the system, co-simulation can be used. In this approach, each component model is simulated with a formalism-specific simulator. Interaction due to component coupling is resolved at the trajectory (simulation data) level. The co-simulation engine orchestrates the flow of input/output data. In this approach, questions about the overall system can only be answered at the level of input/output (state trajectory) level. It is no longer possible to answer higher-level questions which could be answered within the individual components' formalisms. Furthermore, there are speed and numerical accuracy problems for continuous formalisms, in particular if one attempts to support computationally non-causal models. The co-simulation approach is meaningful mostly for discrete-event formalisms. It is the basis of the DoD High Level Architecture (HLA) [14] for simulator interoperability.
- As in co-simulation, each system component may be modelled using the most appropriate formalism and tool. In multi-formalism modelling and simulation however, a single formalism is identified into which each of the component models may be symbolically transformed [23]. Obviously, the system properties which we wish to investigate must be invariant under the transformations. The formalism to transform to depends on the question to be answered about the system. The Formalism Transformation Graph (see Figure 1) suggests DEVS [25] as a universal common modelling formalism for simulation purposes (generating input/output trajectories).

It is easily seen how multi-formalism modelling subsumes both the super-formalism approach and the co-simulation approach.

Although the model transformation approach is conceptually appealing, there remains the difficulty of interconnecting a plethora of different tools, each designed for a particular formalism. Also, it is desirable to have problem-specific formalisms and tools. The time needed to develop these is usually prohibitive. This is why we introduce meta-modelling whereby the different formalisms themselves as well as the transformations between them are modelled. This preempts the problem of tool incompatibility. Ideally, a meta-modelling environment must be able to generate customized tools for models in various formalisms provided the formalisms are described at the meta-model level. When these tools rely on a common data structure to internally represent the models, transformation between formalisms is reduced to the transformation of these data structure.

In this article, we present ATOM^3, a tool which implements the ideas presented above. ATOM^3 has a meta-modelling layer in which different formalisms are modelled graphically. From the meta-specification (in the Entity Relationship formalism), 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. Although graph grammars [5] have been used in very diverse areas such as graphical editors, code optimization, computer architecture, etc. [7], to our knowledge, they have never been applied to formalism transformations.

2 Multi-formalism modelling and the Formalism Transformation Graph

Complex systems are characterized not only by a large number of components, but above all by the diversity of these components (and the feedback interaction between them). For the analysis and design of such complex systems, it is not sufficient to study the individual components in isolation. Properties of the system must be assessed by looking at the *whole* multi-formalism system.

In figure 1, a part of the "formalism space" is depicted in the form of a *Formalism Transformation Graph* (FTG). The different formalisms are shown as nodes in the graph. The arrows denote a homomorphic relationship "can be mapped onto", using symbolic transformations between formalisms. The vertical dashed line is a division between continuous and discrete formalisms. The vertical, dotted arrows denote the existence of a solver (simulation kernel) capable of simulating a model.

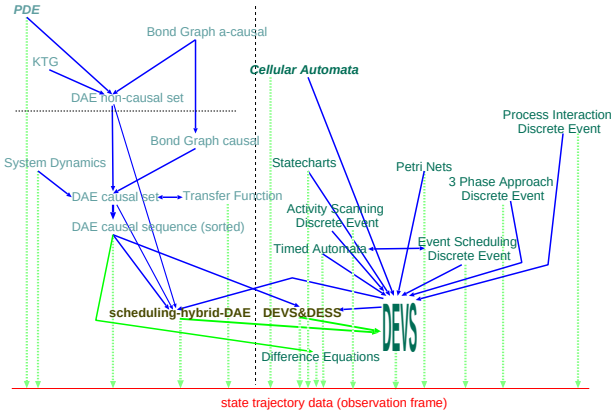

Fig. 1. Formalism Transformation Graph.

3 Meta-Modelling

As stated in the previous section, one of the characteristics of complex systems is the diversity of their components. Consequently, it is often desirable to model the different components using different modelling formalisms. This is certainly

the case, when inter-disciplinary teams collaborate on the development of a single system. A proven method to achieve the required flexibility for a modelling language that supports many formalisms and modelling paradigms is to model the modelling language itself [4] [22]. Such a model of the modelling language is called a meta-model. It describes the possible structures which can be expressed in the language. A meta-model can easily be tailored to specific needs of particular domains. This requires the meta-model modelling formalism to be rich enough to support the constructs needed to define a modelling language. Taking the methodology one step further, the meta-modelling formalism itself may be modelled by means of a meta-meta-model. This meta-meta-model specification captures the basic elements needed to design a formalism. Table 1 depicts the levels considered in our meta-modelling approach. Formalisms such as the ERD

Level	**Description**	**Example**
Meta-Meta-Model	Model used to specify modelling languages	Entity-Relationship Diagrams, UML class Diagrams, etc.
Meta-Model	Model used to specify simulation models	Deterministic Finite Automata, Ordinary differential equations (ODE), etc.
Model	The description of an object in a certain formalism	$f'(x) = -\sin x, f(0) = 0$ (in the ODEs formalism)

Table 1. Meta-modelling levels.

are often used for meta-modelling. 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 (DFA), different transitions leaving a given state must have different labels. This cannot be expressed within ERD's 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 [12], including ATOM^3 use the Object Constraint Language OCL [19] used in the UML.

Figure 2 depicts the structure we propose for a meta-modelling environment. ATOM^3 was initialized using a hand-coded Entity-Relationship (ER) meta-meta-model. As the ER formalism can be described in an ER model, the environment could be bootstrapped.

4 Graph Grammars

In analogy to string grammars, graph grammars can be used to describe graph transformations, or to generate sets of valid graphs. Graph grammars are com-

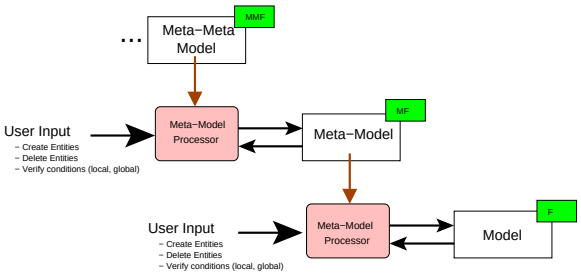

Fig. 2. Proposed working scheme for a meta-modelling environment.

posed of rules, each mapping a graph on the left-hand side (LHS) to a graph on the right-hand side (RHS). When a match is found between the LHS of a rule and a part of an input graph (called host graph), this subgraph is replaced by the RHS of the rule. Rules may also have a condition that must be satisfied in order for the rule to be applied, as well as actions to be performed when the rule is executed. A rewriting system iteratively applies matching rules in the grammar to the graph, until no more rules are applicable.

The use of a model (in the form of a graph grammar) of graph transformations has some advantages over an implicit representation (embedding the transformation computation in a program) [3]:

- It is an abstract, declarative, high level representation.
- The theoretical foundations of graph rewriting systems may assist in proving correctness and convergence properties of the transformation tool.

On the other hand, the use of graph grammars is constrained by efficiency. In the most general case, subgraph isomorphism testing is NP-complete. However, the use of small subgraphs on the LHS of graph grammar rules, as well as using node labels and edge labels can greatly reduce the search space.

Since we store simulation models as graphs, it is possible to express the transformations shown in the FTG as graph grammars at the meta-level.

For example, suppose we want to transform Non-deterministic Finite Automata (NFA) into (behaviourally) equivalent DFA. In the latter formalism, the labels of all transitions leaving a state must be different. Models in both formalisms can be represented as graphs. Figure 3 shows the NFA to DFA transformation specification in the form of a graph grammar.

In this graph grammar, entities are labeled with numbers. In our case, entities are both states and transitions. RHS node labels have also a prima, to be distinguished from LHS ones. If two nodes in a LHS and a RHS have the same number, that means that the node must not desappear when the rule is executed. If a number appears in a LHS but not in a RHS, that means that the node must be removed when applying the rule. If a number appears in a RHS but not in a LHS, that means that the node must be created if the rule is applied.

For subnode matching purposes, we should specify the value of the attributes of the nodes in the LHS that will produce a matching. In the example, all the attributes in LHS nodes have the value of $\langle ANY \rangle$, that means that any value will produce a matching. It is also needed to specify the value of the attributes once the rule has been applied and the LHS has been replaced by the RHS. This is done by specifying attributes in the RHS nodes. If no value is specified, and the node is not a new node (the label appears in the LHS), by default it will keep its values. It is also possible to calculate new values for attributes, and we certainly must do this if a new node is generated when replacing the LHS by the RHS. In the example, we specify new values in nodes 5' and 6' of rules 3 and 4.

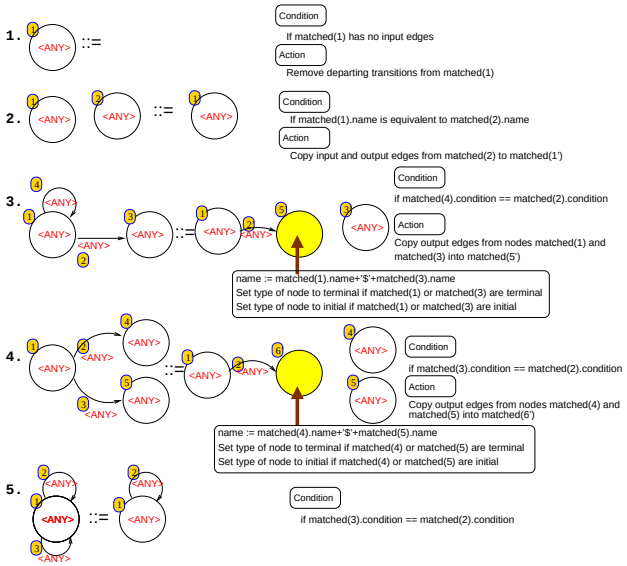

Fig. 3. A grammar to transform NFA into DFA.

In the picture, *matched(i)* means the node in the host graph that makes a match with node i in the rule. The graph grammar rules do the following: Rule one removes unreachable nodes; rule two joins two equal states into one; rule three eliminates non determinism when there are two transitions with the same label departing from the same node, and one goes to a different node while the other goes into the first one; rule four is very similar to the previous one, but the non determinism is between two different nodes; finally, the last rule removes transitions with the same label departing from and arriving to the same state.

A graph rewriting module for formalism transformation takes as inputs a grammar and a model in a source formalism and outputs a behaviourally equivalent model expressed in a target formalism. In some cases, the output and the input models are expressed in the same formalism, and the application of the

graph grammar merely optimizes some aspect the model. A typical example is *constant folding* of Algebraic models. In this case, one of the rules states that the sub-graph consisting of an operator node whose operand nodes are all constant must be replaced by a constant node whose value is the result of applying the operation on the constants. Other uses we make of graph-grammars will be commented in section 5.4.

5 ATOM^3

ATOM^3 is a tool written in Python [21] which uses and implements the concepts presented above. Its architecture is shown in figure 4, and will be explained in the following sections.

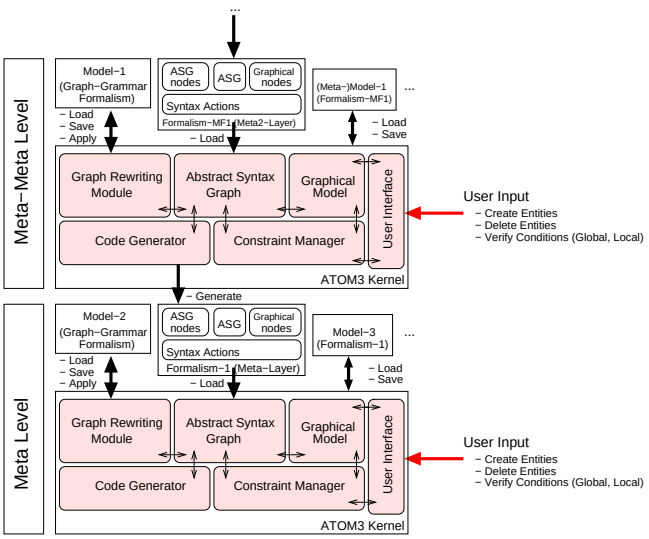

Fig. 4. The ATOM^3 architecture.

The main component of ATOM^3 is the Kernel. This module is responsible for loading, saving, creating and manipulating models, as well as for generating code. By default, a meta-meta-model is loaded when ATOM^3 is invoked. This meta-meta-model allows to model meta-models (modelling formalisms) using a graphical notation. For the moment, the ER formalism extended with constraints is available at the meta-meta-level. 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 DFA Formalism, it is necessary to define both States and Transitions. Furthermore, for States we need to add the attribute name and type (initial, ter-

minal or regular). For Transitions, we need to specify the condition that triggers it.

In general, in ATOM^3 we have two kinds of attributes: *regular* and *generative*. Regular attributes are used to identify characteristics of the current entity. Generative attributes are used to generate new attributes at a lower meta-level. The generated attributes may be generative in their own right. Both types of attributes may contain data or code for pre and post conditions. Thus, in our approach, we can have an arbitrary number of meta-levels as, starting at one level, it is possible to produce a generative attribute at the lower meta-level and so on. The meta-chain ends when a model has no more generative attributes. Attributes can be associated with individual model entities as well as with a model as a whole.

Many modelling formalisms support some form of coupled or network models. In this case, we need to connect entities and to specify restrictions on these connections. In our DFA example, States can be connected to Transitions, although this is not mandatory. Transitions can also be connected to States, although there may be States without incoming Transitions. In ATOM^3, in principle, all objects can be connected to all objects. Usually, a meta-meta-model is used to specify/generate constraints on these connections. Using an ER meta-meta-model, we can specify cardinality constraints in the relationships. These relationships will generate constraints on object connection at the lower meta-level.

The above definitions are used by the Kernel to generate the Abstract Syntax Graph nodes. These nodes are Python classes generated using the information at the meta-meta-level. The Kernel will generate a class for each entity defined in the semantic space and another class for the Abstract Syntax Graph. This class is responsible for storing the nodes of the graph. As we will see later, it also stores global constraints. In the meta-meta-model, it is also possible to specify the graphical appearance of each entity of the lower meta-level. This appearance is, in fact, a special kind of generative attribute. For example, for the DFA, we can choose to represent States as circles with the state's name inside the circle, and Transitions as arrows with the condition on top. That is, we can specify how some semantic attributes are displayed graphically. We must also specify connectors, that is, places where we can link the graphic entities. For example, in Transitions we will specify connectors on both extremes of the arc and in States on 4 symmetric points around the circle. Further on, connection between entities is restricted by the specified semantic constraints. For example, a Transition must be connected to two States. The meta-meta-model generates a Python class for each graphical entity. Thus, semantic and graphical information are separated, although, to be able to access the semantic attributes' values both types of classes (semantic and graphical) have a link to each other.

In the following, we will explore some of the ATOM^3 features in more detail.

5.1 Constraints and Actions

It is possible to specify constraints in both the semantic and the graphical space:

- In the semantic space, it is not always possible to express restrictions by means of class or entity relationship diagrams. For example, in DFA's, we would like to require unique State names, as well as a unique initial State and one or more terminal States. Furthermore, transitions departing from the same State must have different labels.
- In the graphical space, it is often desirable to have the entities' graphical representation change depending on semantic or graphical events or conditions. For example, we would like the representation of States to be different depending on the States' type (initial, regular or terminal).

Constraints can be *local* or *global*. Local constraints are specified on single entities and only involve local attribute values. In global constraints, information about all the entities in a model may be used. In our example, the semantic constraints mentioned before must be specified as global, whereas the graphical constraint is local, as it only involves attributes specific to the entity (the type of the State).

When declaring semantic constraints, it is necessary to specify which event will trigger the evaluation of the constraint, and whether evaluation must take place after (post-condition) or before (pre-condition) the event. The events with which these constraints can associated can be *semantic*, such as saving a model, connecting, creating or deleting two entities, etc., or purely *graphical*, such as moving or selecting an entity, etc. If a pre-condition for an event fails, the event is not executed. If a post-condition for an event fails, the event is undone. Both semantic and graphical constraints can be placed on any kind of event (semantic or graphical). Semantic constraints can be specified as Python functions, or as OCL expressions. In the latter case, they must be translated into Python. Local constraints are incorporated in semantic and graphical classes, global constraints are incorporated in the Abstract Syntax Graph class. In both cases, constraints are encoded as class methods.

When modelling in the ER formalism, the relationships defined between entities in the semantic space create constraints: the types of connected entities must be checked as well as the cardinality of the relationships. The latter constraint may however not be satisfied during the whole modelling process. For example, if we specify that a certain entity must be connected to exactly two entities of another type, at some point in the modelling process the entity can be connected to zero, one, two or more entities. If it is connected to zero or one, an error will be raised only when the model is saved, whereas if it is connected to three or more entities the error can be raised immediatelly. It is envisioned that this evolution of the formalism during the modelling life-cycle will eventually be specified using a variable-structure meta-model (such as a DFA with ER states).

Actions are similar to Constraints, but unlike constraints, actions have side-effects. Actions are currently specified using Python only.

Graphical constraints and actions are very similar to the semantic ones, but they act on graphical attributes.

5.2 Types

In ATOM^3, attributes defined on entities must have a type. All types inherit from an abstract class named *ATOM3Type* and must provide methods to: display a graphical widget to edit the entity's value, check the value's validity, clone itself, make itself persistent, etc.

As stated before, ATOM^3 has two kinds of *basic* types: regular (such as integers, floats, strings, lists of some types, enumerate types, etc) and generative (used to generate an attribute, constraint or graphical attribute at the lower meta-level). There are four types of generative attributes:

1. *ATOM3Attribute*: are used to create attributes at the lower meta-level.
2. *ATOM3Constraint*: are used to create a constraint at the lower meta-level. The code can be expressed in Python or OCL, and the constraint must be associated to some (semantic or graphical) event(s), and must be specified wether it must be evaluated after or just before the event takes place.
3. *ATOM3Appearance*: associate a graphical appearance with the entity at the lower meta-level. Models (as opposed to entities) can also have an associated graphical appearance. This is useful for hierarchical modelling, as models may be inserted inside other models as icons.
4. *ATOM3Cardinality*: are used to generate cardinality constraints on the number of elements connected, at the lower meta-level.

It is also possible to specify *composite* types. These are defined by constructing a type graph [2]. The Meta-model for this graph has been built using ATOM^3 and then incorporated into the Kernel. The components of this graph can be basic or composite types and can be combined using the product and union type operators. Types may be recursively defined, meaning that one of the operands of a product or union operator can be an ancestor node.

Infinite recursive loops are detected using a global constraint in the type meta-model. The graph describing the type is compiled into Python code using a graph grammar (also defined using ATOM^3).

5.3 Code generation

If a model contains generative attributes, ATOM^3 is able to generate a tool to process models defined by the meta-information. "Processing" means constructing models and verifying that such models are valid, although further processing actions can be specified by means of graph grammars. These generated tools also use the Kernel and are composed of:

- The Python classes correspoding to the entities defined in the semantic space. These classes hold semantic information about the attributes, and local constraints (both defined by means of generative attributes in an higher meta-level).
- A Python class used to construct the Abstract Syntax Graph. It holds the global constraints and a dictionary used to store a list of the nodes in the

graph, classified by type. This is useful as operations, such as constraint evaluation can be performed using the *visitor pattern* [10], and the graph can hence be traversed more efficiently.
- Several Python classes to describe the graphical appearance. These classes can have references to semantic attributes, and may also have information about graphical constraints.
- Several Python methods stored in a single file. These methods are added dynamically to the Kernel class. These methods create buttons and menus that allow the creation of new entities, their editing, connection, deletion, etc.

Models are Python functions that contain the executable statements to instantiate the appropriate semantic classes, Abstract Syntax Graph class and graphical classes. In fact, when these statements are executed, the result is identical to the case where the model is constructed interactively by means of the graphical editor. Thus, if one changes models by hand, making them violate some constraint, the Kernel will detect this and react accordingly.

Currently we have implemented the ER formalism at the meta-meta-level. Basically, there are two types of entities: *Entities* and *Relationships*. *Entities* are composed of a name (the keyword), a list of *ATOM3Attribute*, a list of *ATOM3Constraint* and an attribute of type *ATOM3Appearance*. *Relationships*, in addition to the above, have a list of *ATOM3Cardinality* which is filled by means of Post-Actions when an *Entity* is connected to the *Relationship*. By means of pre and post conditions, it is ensured that *Entities* can only be connected to *Relationships*, that the names of *Entities* and *Relationships* are unique, etc. With this meta-meta-model it is possible to define other meta-meta-models, such as class diagrams as inheritance relationships between classes can be implemented with pre and post actions. Note how such an implementation allows for the implementation of various inheritance semantics. Furthermore, target code can be generated in languages (such as C) which do not support inheritance.

Figure 5 shows an example of the ER meta-meta-model in action to describe the DFA Formalism (left side in the picture). This information is used to automatically generate a tool to process DFA models (right side in the picture). On both sides, a dialog box to edit entities is shown. On the right side, the entity that is being edited is a DFA State, that has a name (string) and a type (enumerate type). On the left side, the appearance attribute of an Entity is being edited.

5.4 Formalism Transformation

Once a model is loaded, it is possible to transform it into an equivalent model expressed in another formalism provided the transformations between formalism has been defined. Because the models are expressed internally as graphs, the transformation between formalism can be specified as graph grammars [5].

In ATOM^3, Graph Grammar rules can be modelled as Entities composed of a LHS and a RHS , conditions that must hold for the rule to be applicable and

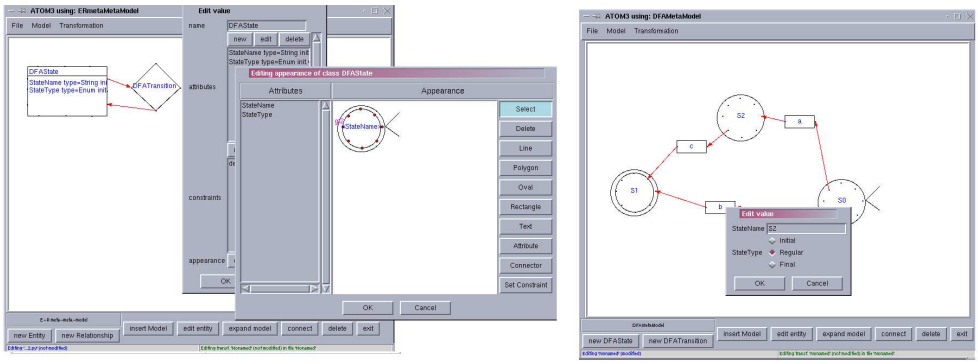

Fig. 5. An example: Generating a tool to process Deterministic Finite Automata.

some actions to be performed when embedding the RHS in the graph. LHS and RHS are indeed meta-models, and may be of different kinds. In figure 3, LHS's are expressed in the NFA formalism, whereas RHS's are expressed in the DFA formalism. For other cases, we can have a mixture of formalisms in both LHS's and RHS's. For this purpose, we allow to open several meta-models at a time. The graph rewriting module uses an improvement of the algorithm described in [5], in which we allow non-connected graphs be part of LHS in rules. It is also possible to define a sequence of graph-grammars that have 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. Control the execution of the rules (stopping after each rule execution or continuous execution) is also possible.

Figure 6 shows a moment in the edition of the LHS of rule 4 of the graph grammar of figure 3. It can be noted that the dialogs to edit the entites have some more fields when these entities are inside a graph grammar rule, namely, the node label and the widgets to set the attribute value to $\langle ANY \rangle$. RHS nodes have extra widgets to copy attribute values from LHS nodes, and to specify their value by means of Python functions.

Besides formalism transformations, we use graph-grammars for other purposes:

- Code generation: We use a graph-grammar to generate Python code for ATOM^3 composite types.
- Simulation: It is possible to describe the operational semantics of models by means of graph-grammars, in particular, we have described a simulator for block diagrams in this way.
- Optimization of models: For example, we have defined a graph-grammar to simplify Structure Charts diagrams (SC's). We usually use this transformation coupled with a graph-grammar to transform Data Flow Diagrams (DFD's) into SC's.

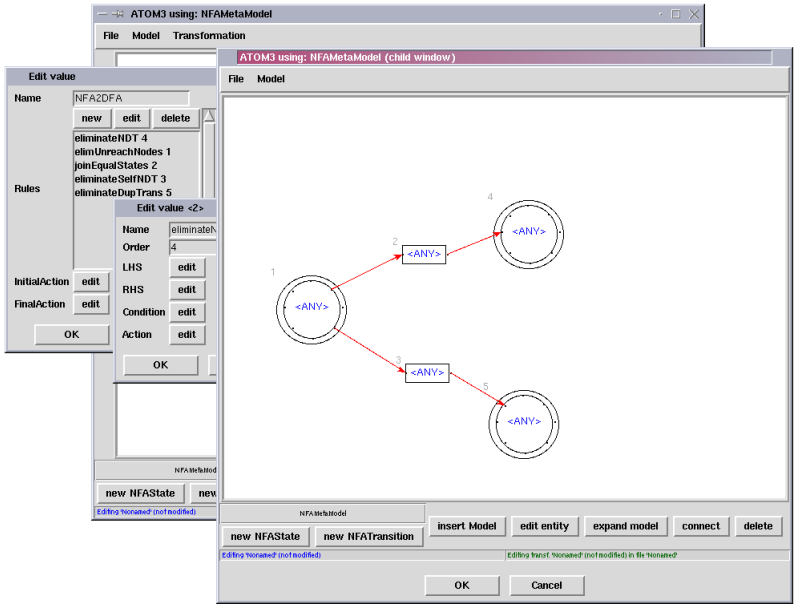

Fig. 6. Editing LHS of rule 4 of the graph grammar in figure 3

6 Related work

A similar approach is ViewPoint Oriented Software Development [9]. Some of the concepts introduced by the authors have a clear counterpart in our aproach (for example, *ViewPoint templates* have an equivalence with meta-models, etc). They also introduce the relationships between ViewPoints, which in our case have an equivalence with coupling of models and graph transformations.

Although this approach has some characteristics that our approach lacks (such as the work plan axioms), our use of graph transformations allows to express model's behaviour and formalism's semantics. These graph transformations allow us to transform models between formalisms, optimize models, or describe basic simulators. Another advantage of our approach, is that we consider Meta-levels, in this way we don't need different tools to process different formalisms (ViewPoints), as we can model them at the meta-level.

Other approaches taken to interconnecting formalism are Category Theory [8], in which formalisms are cast as categories and their relationships as functors. See also [24] and [18] for other approaches.

There are other visual meta-modelling tools, among them DOME [4], Multigraph [22], MetaEdit+ [16] or KOGGE [6]. Some of them allow to express formalism' semantics by means of some kind of textual language (for example, KOGGE uses something similar to Modula-2). Our approach is quite different, because we express such semantics by means of graph grammars. We think that graph-

grammars is a natural and general way to manipulate graphs, rather than using a purely textual language. Some of the rationale for using graph-grammars in our approach was show in section 4. Also, none of the tools consider the possibility to "translate" models between different formalisms.

On the other hand, there are some systems and languages for graph-grammar manipulations, such as PROGRES [20], GRACE [11], AGG [1], etc., although all of them lack of a Meta-Modelling layer.

Our approach is original in the sense that we take the advantages of Meta-Modelling (to avoid explicit programming of customized tools) and graph transformation systems (to express model's behaviour and formalism transformation). The main contribution is thus in the field of multi-paradigm modelling [23] as we have a general means to transform models between different formalisms.

7 Conclusions and future work

In this article, we have presented a new approach for modelling complex systems. Our approach is based on Meta-Modelling and Multi-Formalism modelling, and is implemented in ATOM^3. This code-generating tool, developed in Python, relies on graph grammars and meta-modelling techniques and supports hierarchical modelling.

The advantages of using such 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 kinds 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. Not only we describe formalisms commonly used in the simulation of dynamical systems, but we have also described formalisms such as DFD's and SC's used for the description of software.

A side effect of our code-generating approach is that some parts of the tool have been built using code generated by itself (bootstrapped): one of the first implemented features of ATOM^3 was the capability to generate code, and extra features were added using code thus generated. An example of this is the dialog to specify composite types: the meta-model for this graph has been specified with ATOM^3, and subsequently Python code was automatically generated.

The way of specifying composite types is very flexible, as types are treated as models, and stored as graphs. This means graph grammars can be constructed to specify operations on types, such as discovering infinite recursion loops in their definition, determining if two types are compatible, performing cast operations, etc.

One of the most obvious uses of modelling (although not the only one) is simulation. For that purpose, we describe the dynamic semantics of the models by means of graph grammars.

In the future, we should also explore the possibility of encoding all the code-generation ability of ATOM^3 (see section 5.3) into graph grammar rules.

Currently, the replacement of the basic internal data structure for representing models (graphs) by the more expressive HiGraphs [13] is under consideration. HiGraphs are more suitable to express and visualize hierarchies (blobs can be inside one or more blobs), they add the concept of orthogonality, and blobs can be connected by means of hyperedges.

We also intend to extend the tool to allow collaborative modelling. For this purpose, we are working on putting the APIs for constructing graphical interfaces in Java (Swing) and Python (Tkinter) at the same level. These developments, together with the possibility to use Python on top of the Java Virtual Machine (e.g., by means of Jython [15]), will allow us to make our tool in applet form accessible through a web browser. 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.

Finally, ATOM^3 is being used to build small projects in a Modelling and Simulation course at the School of Computer Science at McGill University.

References

1. AGG Home page: *http://tfs.cs.tu-berlin.de/agg/*
2. Aho, A.V., Sethi, R., Ullman, J.D. 1986. *Compilers, principles, techniques and tools.* Chapter 6, *Type Checking.* Addison-Wesley.
3. Blonstein, D., Fahmy, H., Grbavec, A.. 1996. *Issues in the Practical Use of Graph Rewriting.* Lecture Notes in Computer Science, Vol. 1073, Springer-Verla, 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.* Lecture Notes in Computer Science, 922. Springer.
6. J. Ebert, R. Sttenbach, I. Uhe *Meta-CASE in Practice: a Case for KOGGE* In A. Olive, J. A. Pastor: Advanced Information Systems Engineering, Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Catalonia, Spain, June 16-20, 1997 LNCS 1250, S. 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, Proceedings.* Lecture Notes in Computer Science, Vol. 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. *ViewPoint Oriented Software Development* Proc, of the Third Int. Workshop on Software Engineering and its Applications, Tolouse, December 1990.

10. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1995.
11. GRACE Home page: *http://www.informatik.uni-bremen.de/theorie/GRACEland/GRACEland.html*
12. Gray J., Bapty T., Neema S. 2000. *Aspectifying Constraints in Model-Integrated Computing*, OOPSLA 2000: Workshop on Advanced Separation of Concerns, Minneapolis, MN, October, 2000.
13. Harel, D. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.
14. HLA Home page: *http://hla.dmso.mil*
15. Jython Home Page: *http://www.jython.org*
16. MetaCase Home Page: *http://www.MetaCase.com/*
17. Mosterman, P. and Vangheluwe, H.. *Computer automated multi paradigm modeling in control system design.* In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 65–70. IEEE Computer Society Press, September 2000. Anchorage, Alaska.
18. Niskier, C., Maibaum, T., Schwabe, D. 1989 *A pluralistic Knowledge Based Approach to Software Specification* 2nd European Software Engineering Conference, LNCS 387, Springer Verlag 1989, pp.:411-423
19. OMG Home Page: *http://www.omg.org*
20. PROGRES home page: *http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html*
21. Python home page: *http://www.python.org*
22. Sztipanovits, J., et al. 1995. *"MULTIGRAPH: An architecture for model-integrated computing"*. In ICECCS'95, pp. 361-368, Ft. Lauderdale, Florida, Nov. 1995.
23. Vangheluwe, H. *DEVS as a common denominator for multi-formalism hybrid systems modelling.* In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134. IEEE Computer Society Press, September 2000. Anchorage, Alaska.
24. Zave, P., Jackson, M. 1993. *Conjunction as Composition* ACM Transactions on Software Engineering and Methodology 2(4), 1993, 371-411.
25. Zeigler, B., Praehofer, H. and Kim, T.G. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press, second edition, 2000.