AToM^3: A Tool for Multi-Formalism and Meta-Modelling

Juan de Lara[1,2] 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 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. This 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 and operational semantcis 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 *Bond Graphs*, *Discrete EVent system Specification* (DEVS) [25], *Entity-Relationship* diagrams (ER) and *State charts*. Several approaches are possible:

1. A single *super-formalism* may be constructed which subsumes all the formalisms needed in the system description. This is neither possible nor meaningful in most cases, although there are some examples of formalisms that

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

2. Each system component may be modelled using the most appropriate formalism and tool. To investigate the overall behaviour of the system, *co-simulation* can be used. In this approach, each component is simulated with a formalism-specific simulator. Interaction due to component coupling is resolved at the trajectory (simulation data) level. It is no longer possible to answer symbolic, higher-level questions that could be answered within the individual components' formalisms.

3. As in co-simulation, each system component may be modelled using the most appropriate formalism and tool. In the *multi-formalism* approach however, a single formalism is identified into which each of the components may be symbolically transformed [23]. 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. 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 explicitly. 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 such a tool relies on a common data structure to internally represent models, irrespective of formalism, transformation between formalisms is reduced to the transformation of these data structures.

In this article, we present AToM^3 [1], a tool which implements the above ideas. 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 internally represented using *Abstract Syntax Graphs* (ASGs). As a consequence, transformations between formalisms are reduced to graph rewriting. Thus, the transformations themselves can be expressed as graph grammar models. Although graph grammars [6] have been used in very diverse areas such as graphical editors, code optimization, computer architecture, etc. [8], to our knowledge, they have never been applied to formalism transformation.

2 Preliminaries

2.1 Multi-Formalism Modelling

For the analysis and design of complex systems, it is not sufficient to study 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.

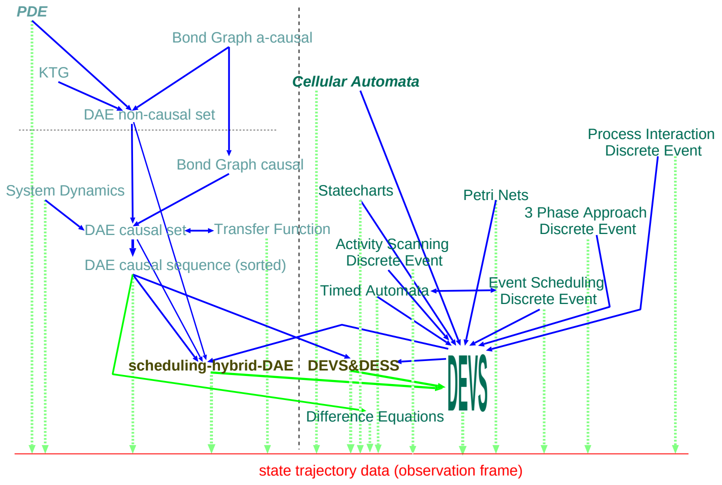


Fig. 1. Formalism Transformation Graph.

2.2 Meta-Modelling

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 [5] [22]. Such a model of the modelling language is called a meta-model. It describes the possible structures that can be expressed in the 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 2.2 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.

Formalisms such as the ER or UML class diagrams [16] are often used for meta-modelling. To be able to fully specify modelling formalisms, the meta-formalism may have to be extended with the ability to express constraints (limiting the number of meaningful models). For example, when modelling a Deterministic Finite Automaton (DFA), different transitions leaving a given state must have different labels. This cannot be expressed within ER 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 [22], including AToM^3 use the Object Constraint Language OCL [19] used in the UML.

2.3 Graph Grammars

In analogy with string grammars, graph grammars can be used to describe graph transformations, or to generate sets of valid graphs. Graph grammars are composed 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, the matching 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. Some approaches also offer control flow specifications. In our tool, rules are ordered based on a user-assigned priority.

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) [4]:

- It is an abstract, declarative, high level representation. This enables exchange, re-use, and symbolic analysis of the transformation model.

- 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 distinct. Models in both formalisms can be represented as graphs. Figure 2 shows the NFA to DFA transformation specification in the form of a graph grammar.

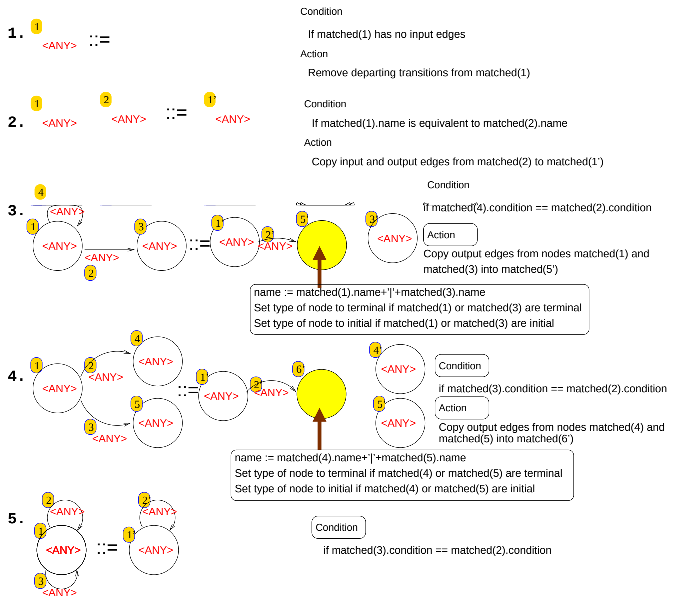
In this graph grammar, entities (both states and transitions) are labelled with numbers. RHS node labels are marked with a prime, to distinguish them from the corresponding LHS ones. If two nodes in a LHS and a RHS have the same number, the node must not disappear when the rule is executed. If a number appears in a LHS but not in a RHS, the node must be removed when applying the rule. If a number appears in a RHS but not in a LHS, the node must be created if the rule is applied.

For subgraph 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 $\langle ANY \rangle$, which means that any value will produce a matching. If a LHS matches, then the additional textual condition (if any) is evaluated. This condition can be specified in Python or in OCL. If this condition holds, the rule can be applied.

It is also necessary 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 respectively.

In the figure, *matched(i)* means "the node in the host graph that matches 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 at 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



1.
<ANY>
::=
Condition
If matched(1) has no input edges
Action
Remove departing transitions from matched(1)
2.
<ANY>
<ANY>
::=
<ANY>
Condition
If matched(1).name is equivalent to matched(2).name
Action
Copy input and output edges from matched(2) to matched(1')
3.
Condition
If matched(4).condition == matched(2).condition
Action
Copy output edges from nodes matched(1) and matched(3) into matched(5')
name := matched(1).name+'|'+matched(3).name
Set type of node to terminal if matched(1) or matched(3) are terminal
Set type of node to initial if matched(1) or matched(3) are initial
4.
Condition
if matched(3).condition == matched(2).condition
Action
Copy output edges from nodes matched(4) and matched(5) into matched(6')
name := matched(4).name+'|'+matched(5).name
Set type of node to terminal if matched(4) or matched(5) are terminal
Set type of node to initial if matched(4) or matched(5) are initial
5.
Condition
if matched(3).condition == matched(2).condition

Fig. 2. A Graph-Grammar to Transform NFA into DFA.

the graph grammar merely optimizes some aspect of the model. Other uses of graph-grammars will be described in section 3.4.

3 AToM^3

AToM^3 is a tool which uses and implements the concepts presented above written in the object-oriented, dynamically typed, interpreted language Python [21]. Its architecture is shown in figure 3, and will be explained in the following sections.

The main component of AToM^3 is the *Processor*, which 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 one 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 attributes *name* and *type* (initial, terminal or regular). For *Transitions*, we need to specify the *condition* that triggers it.

AToM^3 distinguishes between two kinds of attributes: *regular* and *generative*. *Regular* attributes are used to identify characteristics of the current entity.

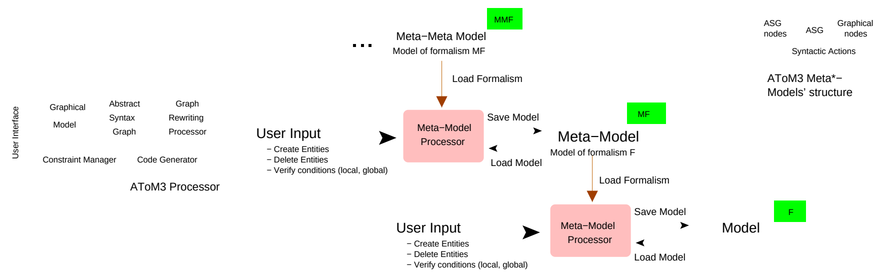


Fig. 3. Meta-... Modelling in ATOM^3.

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-formalisms as, starting at one level, it is possible to produce *generative* attributes 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 (local) as well as with a model as a whole (global).

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 specification is used by the AToM^3 Processor to generate the ASG nodes. These nodes are Python classes generated using the information at the meta-meta-level. The AToM^3 Processor will generate a class for each entity defined in the semantic space and another class for the ASG. 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, connections between entities are 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 one another.

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

3.1 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 ER 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.

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 are associated can be *semantic*, such as saving a model, connecting, creating or deleting 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 types of 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 are translated into Python. *Local* constraints are incorporated in semantic and graphical classes, *global* constraints are incorporated in the ASG 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 immediately. 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 they have side-effects and are currently specified using Python only.

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

3.2 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*. There are four types of *generative* attributes:

1. *ATOM3Attribute*: creates attributes at the lower meta-level.
2. *ATOM3Constraint*: creates 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). It must be specified whether the constraint must be evaluated before or after the event takes place.
3. *ATOM3Appearance*: associates 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 displayed inside other models as icons.
4. *ATOM3Cardinality*: generates 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 [3]. The meta-model for this graph has been built using AToM^3 and then incorporated into the AToM^3 Processor. 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 an 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).

3.3 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 AToM^3 Processor and are composed of:

- The Python classes corresponding 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 a higher meta-level).
- A Python class used to construct the ASG. 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* [12], 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 AToM^3 Processor class. These methods create buttons and menus that allow the creation of new entities, their editing, connection, deletion, etc.

Models are stored as Python functions that contain the executable statements to instantiate the appropriate semantic and graphical classes and the ASG class. 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 edits the generated Python code by hand, making it violate some constraint, the AToM^3 Processor will detect this and react accordingly when such models are loaded.

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 UML 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 4 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.* On the left side, the appearance attribute of an *Entity* is being edited.

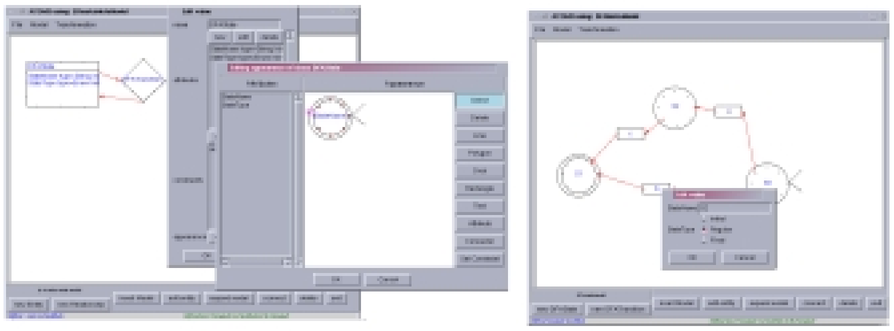


Fig. 4. An Example: Generating a Tool to Process DFA Models.

3.4 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 formalisms has been defined. These transformations can be specified as graph grammar[6] models.

In AToM^3, graph grammar rules are entities composed of a LHS and a RHS, *conditions* that must hold for the rule to be applicable, some *actions* to be performed when embedding the RHS in the graph and a *priority*. LHS and RHS are models, and can be specified within different formalisms. In figure 2, LHSs are expressed in the NFA formalism, whereas RHSs are expressed in the DFA formalism. For other cases, we can have a mixture of formalisms in both LHS and RHS. For this purpose, we allow opening several meta-models at a time.

Graph grammars are entities composed of a list of rules, an initial action and a final action. The graph rewriting processor orders the rules by priority (lower number first) and iteratively applies them to the input graph until none can be applied. After a rule is applied, the first rule of the list is tried again. The graph rewriting processor uses an improvement of the algorithm described in [6], in which we allow non-connected graphs to be part of the 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 model optimization. Rule execution can either be continuous (no user interaction) or step-by-step whereby the user is prompted after each rule execution.

Figure 5 shows a moment in the editing of the LHS of rule 4 of the graph grammar of figure 2. It can be noted that the dialogs to edit the entities have some more fields when these entities are inside the LHS of a graph grammar rule. In particular, 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.

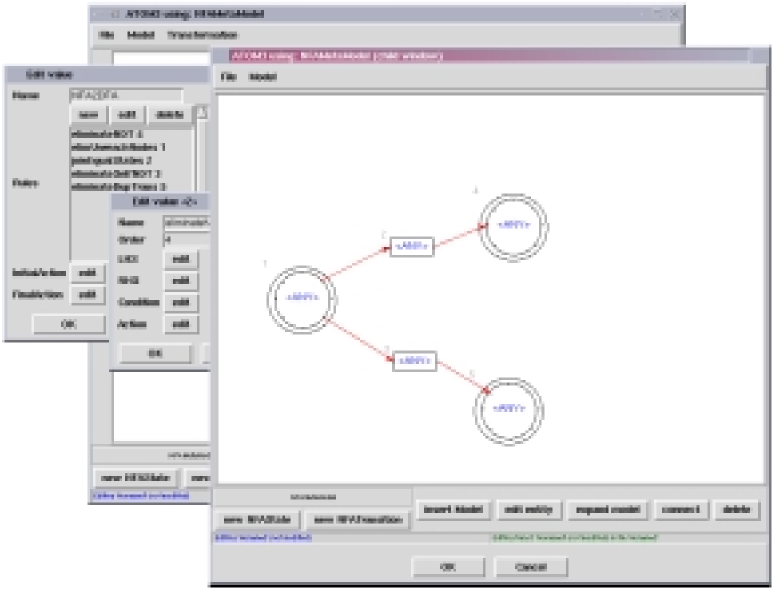


Fig. 5. Editing the LHS of Rule 4 of the Graph-Grammar in Figure 2

Apart from formalism transformation, we use graph-grammars for:

- Code generation: graph grammar rules can control the way the ASG is traversed. For example, we use a graph grammar to generate Python code for AToM^3 composite types. Other examples can be found at the AToM^3 web page [1].
- Simulation: it is possible to describe the operational semantics of models by means of graph-grammars. 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 (SC) diagrams. We usually use this transformation coupled with a graph-grammar to transform Data Flow Diagrams into SC.

4 Related work

A similar approach is ViewPoint Oriented Software Development [11]. Some of the concepts introduced by the authors have a clear counterpart in our approach (for example, *ViewPoint templates* are similar to meta-models). They also introduce the relationships between ViewPoints, which are similar to our 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 use meta-modelling, in this way we don't need different tools to process different formalisms (ViewPoints), as we can model them at the meta-level. See also [9] for an approach to integrate heterogeneous specifications of the same system using graph grammars and the ViewPoint framework.

Other approaches taken to interconnecting formalisms are Category Theory [10], 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 [5], Multigraph [22], MetaEdit+ [15] or KOGGE [7]. Some of them allow to express formalism' semantics by means of some kind of textual language (for example, KOGGE uses a Modula-2-like language). Our approach is quite different. We express semantics by means of graph grammar models. We believe graph grammars are 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 shown in section 2.3. Also, none of the tools consider the possibility to transform models between different formalisms.

There are some systems and languages for graph grammar manipulations, such as PROGRES [20], GRACE [13], AGG [2]. None of these have a meta-modelling layer.

Our approach is original in the sense that we take the advantages of meta-modelling (to avoid explicit programming of custom tools) and graph transformation systems (to express model 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.

5 Conclusions and future work

In this article, we have presented a new approach to the modelling of complex systems. Our approach is based on meta-modelling and multi-formalism modelling, and is implemented in the software tool 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 at the meta-level by means of graph grammars. 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 transformation 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.

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.

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 possible drawback of the approach taken in AToM^3 is that even for non-graphical formalisms, one must devise a graphical representation. For example, in the case of *Algebraic Equations*, the equations must be drawn in the form of a graph. To solve this problem, we will add the possibility to enter models textually. This text will be parsed into an ASG. Once the model is in this form, it can be treated as any other (graphical) model.

Currently, the replacement of the basic internal data structure for representing models (graphs) by the more expressive HiGraphs [14] 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. 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 &Simulation course at the School of Computer Science at McGill University. It can be downloaded from [1], where some examples can also be found.

Acknowledgement

This paper has been partially sponsored by the Spanish Interdepartmental Commission of Science and Technology (CICYT), project number TEL1999-0181. Prof. Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant.

References

1. AToM^3 Home page:
http://moncs.cs.mcgill.ca/MSDL/research/projects/ATOM3.html
2. AGG Home page: http://tfs.cs.tu-berlin.de/agg/
3. Aho, A.V., Sethi, R., Ullman, J.D. 1986. Compilers, principles, techniques and tools. Chapter 6, Type Checking. Addison-Wesley.
4. Blonstein, D., Fahmy, H., Grbavec, A.. 1996. Issues in the Practical Use of Graph Rewriting. Lecture Notes in Computer Science, Vol. 1073, Springer, pp.38-55.
5. DOME guide. http://www.htc.honeywell.com/dome/, Honeywell Technology Center. Honeywell, 1999, version 5.2.1

6. Dorr, H. 1995. Efficient Graph Rewriting and its implementation. Lecture Notes in Computer Science, 922. Springer.
7. J. Ebert, R. Sttenbach, I. Uhe 1997. Meta-CASE in Practice: a Case for KOGGE. Advanced Information Systems Engineering, Proceedings of the 9th International Conference, CAiSE'97 LNCS 1250, S. 203-216, Berlin, 1997. See KOGGE home page at: http://www.uni-koblenz.de/~ist/kogge.en.html
8. Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) 1991. Graph Grammars and their application to Computer Science: 4th International Workshop, Proceedings. Lecture Notes in Computer Science, Vol. 532, Springer.
9. Enders, B.E., Heverhagen, T., Goedicke, M., Troepfner, P., Tracht, R. 2001. Towards an Integration of Different Specification Methods by Using the ViewPoint Framework. Special Issue of the Transactions of the SDPS: Journal of Integrated Design&Process Science, Society for Design&Process Science, forthcoming.
10. 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.
11. Finkelstein, A., Kramer, J., Goedickie, M. 1990. ViewPoint Oriented Software Development. Proc, of the Third Int. Workshop on Software Engineering and its Applications.
12. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design Patterns, Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley.
13. GRACE Home page: http://www.informatik.uni-bremen.de/theorie/GRACEland/GRACEland.html
14. Harel, D. 1998. On visual formalisms. Communications of the ACM, 31(5):514–530.
15. MetaCase Home Page: http://www.MetaCase.com/
16. Meta-Modelling Facility, from the precise UML group: http://www.cs.york.ac.uk/puml/mmf/index.html
17. Mosterman, P. and Vangheluwe, H. 2000. Computer automated multi paradigm modeling in control system design. IEEE Symposium on Computer-Aided Control System Design, pp.:65–70. IEEE Computer Society Press.
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. 2000. DEVS as a common denominator for multi-formalism hybrid systems modelling. IEEE Symposium on Computer-Aided Control System Design, pp.:129–134. IEEE Computer Society Press.
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. 2000. Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press, second edition.