Electronic Notes in Theoretical Computer Science 72 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 15 pages

Using Meta-Modelling and Graph Grammars to create Modelling Environments

Juan de Lara Jaramillo [1]

ETS Informática
Universidad Autónoma de Madrid
Madrid, SPAIN

Hans Vangheluwe [2]

School of Computer Science
McGill University
Montreal, CANADA

Manuel Alfonseca Moreno [3]

ETS Informática
Universidad Autónoma de Madrid
Madrid, SPAIN

Abstract

This paper presents the combined use of meta-modelling and graph grammars for the generation of visual modelling tools for simulation formalisms. In meta-modelling, formalisms are described at a meta-level. This information is used by a meta-model processor to generate modelling tools for the described formalisms. We combine meta-modelling with graph grammars to extend the model manipulation capabilities of the generated modelling tools, as we store (meta-)models as graphs, and thus, express model manipulations as graph grammars.

We show the design and implementation of these concepts in AToM^3 (A Tool for Multi-formalism, Meta-Modelling). As an example we will present a meta-model for Causal Block Diagrams and a graph grammar to generate OOCSMP code, a continuous simulation language which has a compiler able to generate Java applets from the simulations models.

[1] Email: Juan.Lara@ii.uam.es
[2] Email: hv@cs.mcgill.ca
[3] Email: Manuel.Alfonseca@ii.uam.es

1 Introduction

Meta-Modelling is the process of modelling formalisms. In the context of Modelling and Simulation we are interested in formalisms such as Petri-Nets, DEVS, Causal Block Diagrams (CBDs) and Ordinary Differential Equations (ODEs).

A model of a formalism should contain enough information to permit the automatic generation of a tool to check and build models subject to the described formalism syntax. The advantage of this meta-modelling approach is clear: instead of building a whole application from scratch, it is only necessary to specify the kind of models we will deal with. If this specification is done graphically, the time to develop a modelling tool can be drastically reduced to a few hours. Other benefits, such as reduction of testing, ease of change and maintainability are also obtained.

At least, the generated tool should be able to allow the construction of valid models and discover errors in their construction. If (meta-)models are stored as graphs, further manipulations of the models can be described as graph grammars [5]. In Modelling and Simulation we are interested in model manipulations such as:

- Model simulation.
- Model optimization, for example, reducing its complexity.
- Model transformation into another (behaviourally equivalent) model, expressed in a different formalism.
- Generation of (textual) code for existing simulators or tools. In this paper we will focus on this application of model transformation.

In this article, we present AToM^3 [4], a tool which implements the ideas presented above. AToM^3 has a meta-modelling layer in which formalisms are modelled graphically. From the meta-specification (a model 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*, a generalization of the concept of *Abstract Syntax Trees* in compilers, which represents – in the form of a graph – the semantic information of the model built by the user. As a consequence, model manipulation can be expressed as graph grammars.

As an example, we will show the generation of a tool to graphically manipulate CBD models. We will also define a graph grammar to generate textual code for the object-oriented continuous simulation language OOCSMP [1]. CBD models are compiled within AToM^3 by invoking the OOCSMP compiler – which is able to produce Java applets – and then executed by launching the Java Virtual Machine. This is the alternative approach to the one taken in [9], in which we defined a graph grammar (with AToM^3) to simulate the model inside AToM^3 itself.

Level	Description	Example
Meta-Meta-Model	Model describes a formalism that will be used to describe other formalisms.	Description of Entity-Relationship Diagrams, UML Class Diagrams
Meta-Model	Model 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.

2 Meta-Modelling

Meta-Modelling is the process of modelling formalisms. Formalisms are described as models using meta-formalisms. The latter are nothing but formalisms expressive enough, such as Entity Relationship diagrams 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 2 depicts the levels considered in our meta-modelling approach. Note that we only consider three levels, although it can be the case that a meta-formalism mf_1 is powerful enough to describe the meta-meta-model of another meta-formalism mf_2. We consider both mf_1 and mf_2 as meta-formalisms and place them in the same meta-level. As we will see later, in AToM^3 it is usually the case that meta-formalisms can describe meta-formalisms as well as formalisms.

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, 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-formalism. Whereas the meta-formalism frequently uses a graphical notation, constraints are concisely expressed in textual form. For this purpose, some systems [6] (including ours) use the Object Constraint Language OCL [8] used in UML. As AToM^3 [2] is implemented in the scripting language Python [10], arbitrary Python code can also be used.

Other alternative to using constraints is to express in a graph grammar the kind of editing actions the user can perform at each moment in the modelling phase. This approach is called *syntax-directed* [3]. Other kind of visual editors are called *free-hand* [7] and allow the user more flexibility in the model editing phase, but they have to check that the model the user is building is correct.

In AToM^3, free-hand editing is the default approach, and model correctness is guarenteed by evaluating the constraints defined at the meta-level (and associated with events) when the user is building her model. In AToM^3, free-hand editing can be combined with the syntax-directed approach by building graph grammar rules for editing tasks. See section 6 for some comments about this.

3 AToM^3: An Overview

AToM^3 is a tool which uses and implements the concepts presented above. As it has been implemented in Python, it is able to run (without any change) in all platforms in which an interpreter for Python is available: Linux, Windows and McIntosh. The main idea of the tool is: *"everything is a model"*, in the sense that, during the implementation, the AToM^3 kernel has been grown from a small initial kernel, models were defined for other parts of it, code was generated and then later incorporated into it. Also, for AToM^3 users, it is possible to modify some of this model-defined components, such as the type system, the (meta-)formalisms, the user interface, etc.

AToM^3's architecture is shown in Figures 1 and 2. In both of them, models are represented as white boxes, having on their upper-right corner an indication of the meta-...model (formalism) they were specified with. In Figure 1, and for the example in this paper, meta-modelling CBD, the meta-meta-model is Entity-Relationship (the *MMF* is also Entity-Relationship, as this meta-formalism was bootstraped) as we used this meta-formalism to describe what the syntax of CBD is. The Meta-Model obtained is thus CBD, the meta-formalism *MF* is Entity-Relationship. Finally, using this CBD meta-model, it is possible to build CBD models such as the one shown in Figure 5, which describes the harmonic movement in one dimension.

In figure 2, it can be seen that, in the case of a graph grammar model, to convert a model from formalism F_{source} to F_{dest}, it is necessary to use the meta-models of both F_{source} and F_{dest}, together with the meta-model for graph-grammars. The graph grammar we are presenting in this paper, which generates OOCSMP textual code from CBD models, is indeed not a model transformation. That is, our source formalism (F_{source} in figure 2) is CBD, but we do not need a meta-model for OOCSMP (F_{dest} in figure 2) as we are directly generating OOCSMP textual code from the CBD model, rather than representing internally the OOCSMP models as *Abstract Syntax Graphs*.

The main component of AToM^3 is the AToM^3 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 customised tools. Both meta-models and meta-meta-models can be loaded into AToM^3 as shown in Figure 1. The first kind of models allows constructing valid models in a certain formalism, the second is used to describe the formalisms themselves.

The Entity-Relationship formalism extended with constraints is available

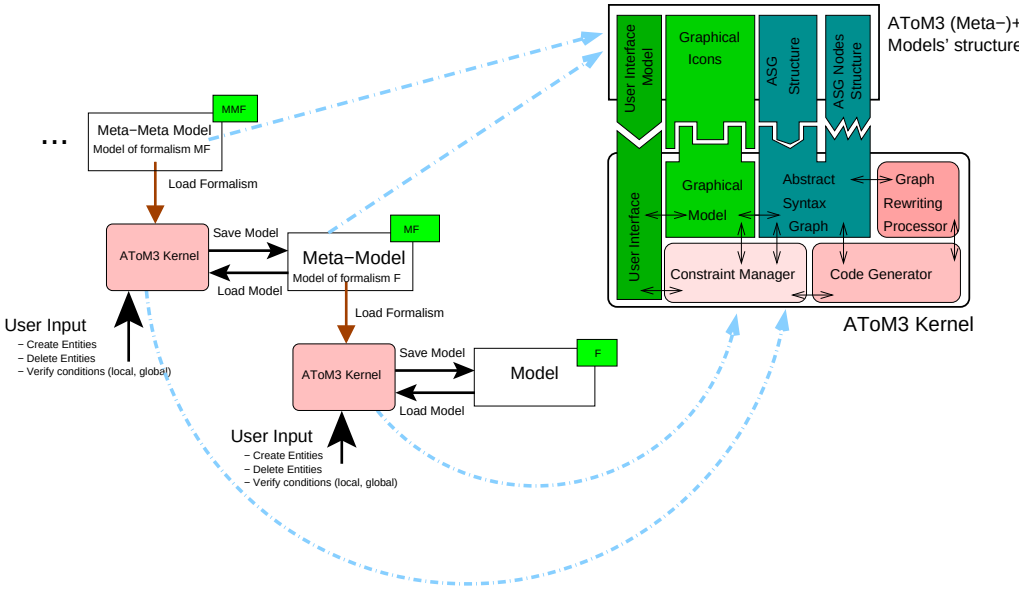

Fig. 1. Meta-... Modelling in AToM^3.

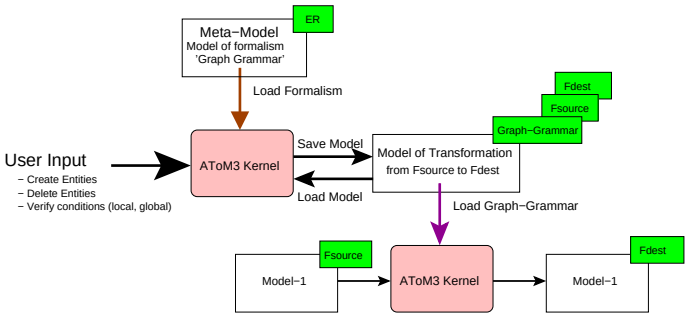

Fig. 2. Model Transformation in AToM^3.

at the meta-meta-level. As stated before, it is perfectly possible to define other meta-formalisms using Entity-Relationship. 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 an attribute, connecting two entities, etc.) or *graphical* (such as dragging, dropping, etc.)

When modelling at the (meta-)$^+$level, the entities that 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 Net 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 the *name*.

The (meta-)$^+$information is used by the AToM^3 Kernel to generate some Python files, which, when loaded by the Kernel, allow the processing of models in the defined formalism (see upper-right corner in Figure 1, labeled as *"AToM^3 (meta-)$^+$models structure"*.)

One of the components of the generated files is a model of a part of the AToM^3 user interface. This User Interface model follows the *"Buttons"* formalism, and has its own meta-model. Initially, this model represents the necessary buttons to create the entities defined in the formalism's meta-model, but can be modified to include, for example, buttons to execute graph grammars on the current model. In the example of this paper, we will define a graph grammar to generate OOCSMP code from the CBD models. In this way, we will add a button to the user interface to execute this graph grammar, invoke the OOCSMP compiler with the generated code, and execute the resulting applets.

In AToM^3, entities may 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.

Entities are connected by means of ports, which can be *named* or *unnamed*. An entity may have both types of ports. Unnamed ports are used when all the connections are semantically equal and we do not need to distinguish them. A typical example is Petri-Nets, in which *places* have unnamed ports to connect to *transitions*. *Named* ports are used when we have different meanings for the same types of connections. A typical example of this is in the CBD formalism, where some entities represent functions to which other entities may be connected, representing the function's parameters. One needs to know exactly which parameter corresponds to each connection. For example, an *INTEGRAL* block has two parameters: the initial condition and the value to be integrated. If we connect a block to an *INTEGRAL*, we need to know if this connection is to be interpreted as the initial condition or as the value. In this way, two named ports are needed for the *INTEGRAL* block to store the connections to each parameter.

In the meta-model, it is also possible to specify the graphical appearance of each entity of the defined formalism. This appearance is, in fact, a special kind of *generative* attribute. Objects' graphical appearance can be icon-like or arrow-like with optional icon decorations in the center, segments and extremes. 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. For connections between *Places* and *Transitions*, we can choose arrow-like appearances with the *weight* shown on top of these arrows. Constraints can also be associated with the graphical entities. Each graphical form, part of the graphical entity, can be referenced by an automatically generated name that has methods to change its colour or hide it.

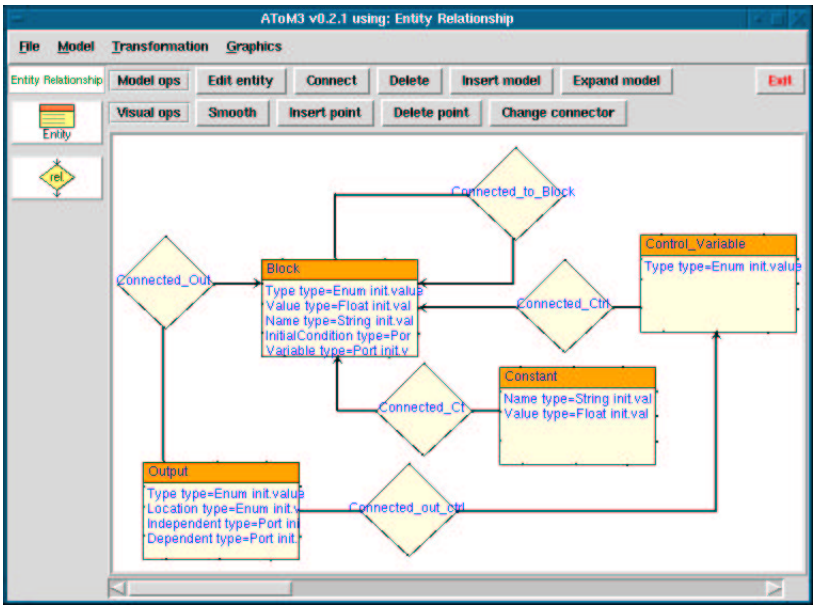

Fig. 3. Meta-model of CBDs, expressed in the ER formalism. (AToM^3 running on Linux)

4 Describing the Causal Block Diagrams Formalism

In AToM^3, we can describe meta-models in the ER formalism. The basic element of the CBD formalism is the *Block*, which represents transfer functions, such as arithmetic operators or integrators. *Blocks* can be connected to other blocks, these connections transmit *signals* between blocks. Signals are functions of time. In the meta-model in figure 3 we have also included entities to represent constants, control variables (such as *TIME*, the basic time interval, etc.), and outputs. These last elements will be connected to the blocks whose values we want to visualize (print or plot) in the simulation execution. Thus, the meta-model is made of the following entities:

- *Block* entities, composed of a field named *Type*, which is an enumerate type that indicates the kind of function this block performs. These functions include infix n-ary operators, such as *"+"* and *"*"*, prefix unary operators, such as *"-"* and *"^1"* (the inverse operator), and functions such as *INTEGRAL*, *DERIVATIVE*, etc. *Block* entities also have a *Value*, which is the result of the application of the block's function to its parameters; a *Name*, which is a *string*, and several named ports to connect its parameters.
- *Constant* entities, which represent values that do not change during the simulation. They are composed of a *Value* (a float) and a *Name* (a string). The *Name* attribute in both *Block* and *Constant* entities is a unique identifier which is automatically generated by AToM^3, although the user can modify it.
- *Output* entities are used to indicate which *Block* values should be displayed. This entity is composed of an attribute named *Type* to select whether the

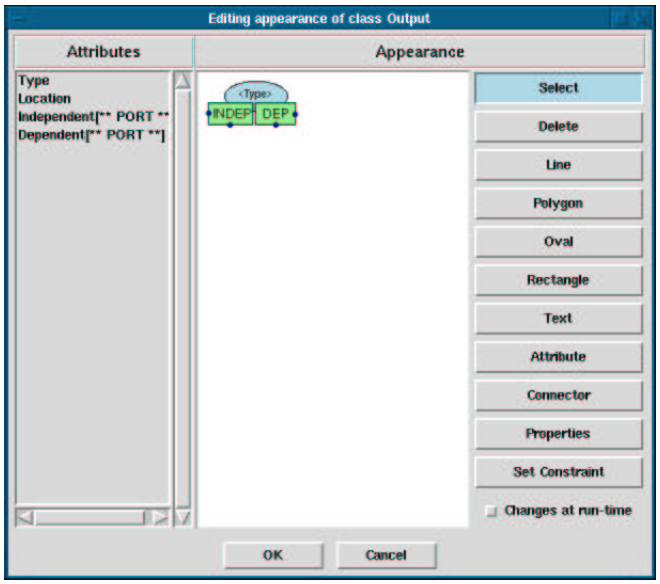

Fig. 4. Defining the graphical appearance of *Output* entities. (AToM^3 running on Linux)

value should to be plotted or printed. They also have another attribute called *Location* to select in which part of the user interface the output panel should be located. It accepts nine possible values: *NW, N, NE, W, C, E, SW, S, SE*. Two additional named ports, *Dependent* and *Independent*, allow us to distinguish between dependent and independent variables while plotting. Figure 4 shows a dialog window used to describe the graphical appearance of *Output* entities. The list on the left shows the semantic attributes of the entity. The canvas in the middle allows the user to draw the graphical appearance that will be associated with the entity. In this canvas, it is possible to show semantic attributes. In this case the canvas shows the *Type* attribute. Notice also that it is possible to put in the canvas as many instances of the named ports (the two last attributes in the *Attributes* list) as desired, but all connections to any of the instances of the same named port are stored in the same semantic attribute. In this example, we have added two instances of ports *Dependent* and *Independent* (shown as little circles in the border of the rectangles). The button labeled *Connector* is used to add unnamed ports to the canvas. Notice also that it is possible to specify graphical constraints by means of the *"Set Constraint"* button.

- *Control_Variable* entities are used to include in the model variables that are used by the simulator to control the simulator execution. These entities have only one attribute, which is an enumerate type with the control variable to be selected: *TIME* (the simulation time), *delta*, (the time step size for the numerical integration) *PRdelta* and *PLdelta* (communication intervals, that is, time elapsed between output refreshes, for printing panels and for plot panels)

In order to keep the model correct, some constraints have to be added:

- The number of connections to the *Independent* port of *Output* entities is one. This constraint is local to *Output* entities and must be verified before saving and before applying a graph grammar. In this way, we ensure that all saved models are correct and that all models to which a graph grammar is applied (such as the one for code generation explained in section 5) are also correct.
- The number of connections that *Block* entities can receive depends on its type. For example, integrators receive two, whereas adders can receive a number of connections greater than one, etc. This means that we cannot set the exact number as the arity of the relationship *Connected_to_Block* in the meta-model. We have to set a local constraint on *Block* entities, that makes sure that depending on the *Type* attribute, the number of connections is the correct one.

With this information, AToM^3 generates some Python files (see figure 1) that, when loaded on top of AToM^3, allow the user to build CBD models. One of these generated Python files is a model of the User Interface that will appear in this CBD tool. The model is expressed in the *"Buttons"* formalism. This formalism allows the user to define the buttons that will appear in a user interface, their text or icon, and the associated action. By default, AToM^3 generates a model with a Button to create each entity and relationship in the meta-model. Of course, this model can be modified. In our case we have modified the model to delete the buttons corresponding to the relationship, and to add a new one to execute a graph grammar to generate OOCSMP code, and to execute the resulting applet after compiling it with the OOCSMP compiler. Additionally, images have been assigned to the buttons.

AToM^3 generates this user interface model by executing a graph grammar (expressed in the Entity Relationship formalism) on the meta-model whose interface is to be generated. The graph grammar traverses the model and converts each entity and each relationship into a *Button* (the basic entity of the *Buttons* formalism). When a formalism is loaded, this user interface model is interpreted by AToM^3 to create the real buttons in the user interface.

Figure 5 shows the generated tool to process CBD models. In this figure we can see a model of the 1D harmonic movement without friction which, for example, governs the movement of a mass attached to a spring (in the absence of friction). The equation describing this movement is:

$$\frac{d^2x}{dt^2} = -\frac{K}{M}x \tag{1}$$

In the CBD formalism, it is possible to directly express this second derivative, but as the simulation will be carried out numerically, the solution will be more accurate if we transform the equation in such a way that only integrals are left, that is, if we integrate twice both sides of the equation. The resulting

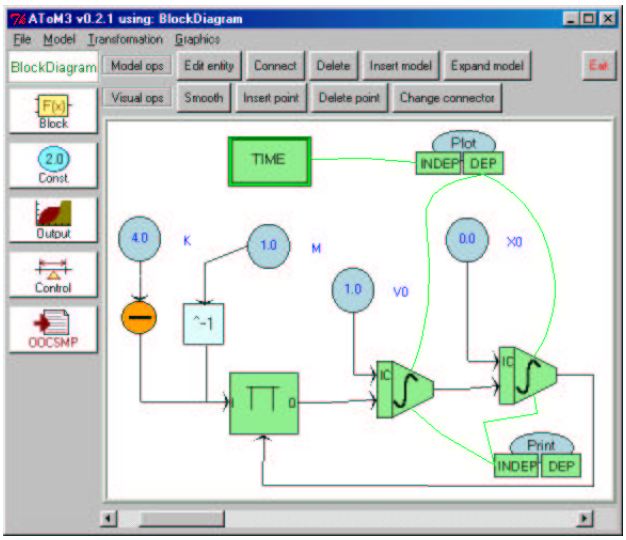

Fig. 5. Generated tool to process CBD models. (AToM^3 running on Windows)

equation is thus:

$$x = x_0 + \int_0^t (v_0 - \frac{K}{M} \int_0^t x dt) dt \tag{2}$$

which is the equation that has been coded in the model in figure 5 (x_0 and v_0 are the initial conditions, initial position and initial velocity respectively). On the left of the model, the block labeled with the product symbol π calculates $-\frac{K}{M}$, where K has been assigned a value of 4.0 and M a value of 1.0 (constants on top of the blocks labeled as "-" and as "^-1"). The other two blocks perform the integrations: in port *"IC"* they receive the initial condition and in the other port the value to be integrated. It must also be noted that the result of these integrals is going to be shown in print and plot panels (see the *Plot* and *Print* blocks).

5 Generating OOCSMP code

OOCSMP [1] is an Object Oriented extension of the CSMP Continuous Simulation Language, developed at the Universidad Autónoma de Madrid. A compiler (called C-OOL) is able to produce Java applets from the OOCSMP models. These applets can be inserted in web documents and placed in the web. One of the main drawbacks of the system is the lack of a graphical modelling environment, models are textual files which must be coded by hand. The work in this paper aims to provide such an environment for a small subset of the OOCSMP syntax.

In OOCMSP models, instructions are indeed equations, which the compiler sorts appropriatelly in order to be able to solve them. Equations are arranged in procedures, the main one called *DYNAMIC*, which is the main section of

the model and gets solved once for each instant of time. In OOCSMP models, one should also specify *control variable* values. These variables control some of the simulator solver parameters, such as the time step, the communication interval, etc. As an example, listing 1 shows the OOCSMP program equivalent to the graphical model in figure 5. It must be noted, that, in truly object-oriented OOCSMP models, definitions and instantiations of classes are also found.

```
TITLE HARMONIC 1D WITHOUT FRICTION
* Author: Juan, Hans, Manuel
DATA K:=4.0
DATA M:=1.0
DATA V0:=1.0
DATA X0:=0.0
DYNAMIC
 MK:=-K
 KBY1M:=MK*1M*X
 1M:=1/M
 V:=INTGRL(V0,KBY1M)
 X:=INTGRL(X0,V)
PLOT [C], V, X, TIME
PRINT [E], V, X
TIMER delta:= 0.1,PLdelta:= 0.1,PRdelta:= 1.0,FINTIM:= 10.0
```

Listing 1: OOCSMP code equivalent to model in figure 5

In this section we show how to use the modelling environment generated in the previous section to generate OOCSMP code. We do this by defining a graph grammar to traverse the CBD model and generate the OOCSMP code. In AToM^3, graph grammars can include actions to be performed before and after the graph grammar execution. In our case, before the graph grammar execution, we open a file to store the OOCSMP code and add an extra attribute (*visited*) to all the nodes of the graph. This attribute controls whether code for that node has been already generated, and is initialized to 0. This initial action also writes the name of the simulation model and the author in the file (see the first two lines in listing 1)

The graph grammar is composed of three rules, none of which changes the matching subgraph:

- Rule number one is applied when a *Constant* node is found that has not been previously processed. The rule generates a *DATA* statement for the constant node and marks it as visited. This rule generates the four data sentences in listing one, for constants *K*, *M*, *V0* and *X0*.
- Rule number two is applied when a *Block* node is found that has not been previously processed. It generates the appropriate OOCSMP syntax, depending on the type of the block, and marks it as visited. The first time it generates a block, it writes the beginning of the *DYNAMIC* section. This

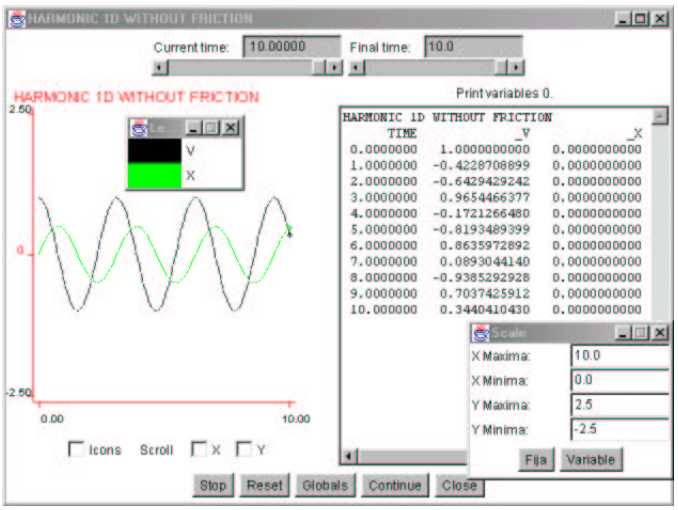

Fig. 6. A moment in the execution of the applet generated by the OOCSMP compiler.

rule generates the six lines after the data declaration section.

- Rule number three is applied when an *Output* node is found that has not been previously processed. It generates the necessary OOCSMP code to output the variables connected to it. Outputs can be plotted or printed. This rule generates the two lines beginning with *PLOT* and *PRINT* in listing 1. We create two panels, one in the center (parameter *"[C]"*) for plotting the V and X variables with respect to *TIME*, and the other to the east (parameter *"[E]"*), to print also the values of variables V and X.

The final action in the grammar generates code to give values to the control variables, including the final time, the communication intervals (*PLdelta* and *PRdelta*) and the time advance (*delta*). These were indicated as global attributes of the model (see last line in listing 1). Finally, we have assigned the execution of this graph grammar to a button (labeled as *OOCSMP* in figure 5. This is done by modifying the user interface model that AToM^3 generates, as mentioned in the previous section.

Figure 6 shows a moment in the execution of the applet generated by the OOCSMP compiler. This corresponds to the model in listing one, which was obtained by applying this graph grammar to the model in figure 5. More information about OOCSMP can be found at:
`http://www.ii.uam.es/~jlara/investigacion`

Of course, there are more efficient ways to generate code from visual models than by using a graph grammar. But they provide high level control mechanisms which allow the user to perform complex manipulations and graph matching, and the user can specify model manipulations without any knowledge of the AToM^3 internals.

6 Related work

There are some other similar tools, such as GenGed [3], which is a tool to build *syntax directed* visual environments. Their ideas are similar to ours, except that we usually do not use a syntax directed approach for the environments we generate, although perfectly possible in AToM^3. We feel that, for some cases, a syntax directed environment is too difficult and restrictive for the final user. We also do not pose the diagram's graphical appearance as Constraint Satisfaction Problems: it is the meta-model designer who, by means of pre- and post- conditions and actions, expressed as Python code, must take care of the graphical layout.

Other tools, such as DiaGen [7] (and AToM^3), may combine characteristics of *free-hand* and *syntax- directed* editors. DiaGen is a tool based on *hypergraph* grammars. The user inputs a textual specification of the visual language and obtains a set of Java classes which are complemented by a Java library (which is part of the OOCSMP distribution) to obtain the visual environment. In AToM^3, the specification of the visual language (the meta-model) is specified graphically, and the generated files are indeed loaded again on top of AToM^3. There is no structural difference between the generated editors (which in fact, could be used to generate other ones!), and the editor which generated them. In fact, one of the main differences with other similar tools of the approach taken in AToM^3, is the concept that (almost) everything in AToM^3 has been defined by a model (under the rules of some formalism), and thus, the user can change it.

7 Conclusions and Future work

In this paper we have presented an overview of AToM^3, a tool which makes the generation of modelling tools possible by combining meta-modelling and graph grammars. By means of meta-modelling, it is easy to define the (graphical) syntax of the kind of models we are interested in. By means of graph grammars we can express model manipulation, such as simulation, optimization, transformation and code generation. As an example, we have presented the generation of a visual modelling environment for a small part of the OOCSMP syntax. For that purpose, we have designed a meta-model for causal block diagrams, and a graph grammar to generate OOCSMP code.

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 (for example, one that already has a

simulator available). 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.

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 at a certain moment of the modelling of a multi-formalism system. This module may assist in deciding which formalism to use to transform each component (following the Formalism Transformation Graph, see [11]).
- 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.

With respect to AToM^3 as a front end for OOCSMP, we would like to improve the graph grammar for code generation, to distinguish expressions whose value is not going to change during the simulation (for example, expressions *MK:=-K* and *1M:=1/M*). These values may be calculated at the begining of the simulation (in a section called *INITIAL*) rather than at each time step, in the *DYNAMIC* section.

We are also working to create a meta-model for UML class diagrams, where methods can be specified as CBD models. This will allow us to generate truly object-oriented OOCSMP models.

Acknowledgements

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. The authors would like to thank three anonymous referees and Simon Lacoste-Julien for their useful comments.

References

[1] Alfonseca,M., Pulido, E., Orosco, R., de Lara, J. 1997. *OOCSMP: An Object-Oriented Simulation Language*. Proceedings of the 9th European Simulation Symposium ESS97, SCS Int., Erlangen, Germany, pp. 44–48. See the OOCSMP home page at:
`http://www.ii.uam.es/~jlara/investigacion/download/OOCSMP.html`.

[2] AToM^3 home page:
`http://moncs.cs.mcgill.ca/MSDL/research/projects/ATOM3.html`

[3] Bardohl, R., 2002 *A Visual Environment for Visual Languages* Science of Computer Programming 44, pp.: 181-203.

[4] de Lara, J., Vangheluwe, H. 2002 *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* In European Conferences on Theory And Practice of Software Engineering ETAPS02, Fundamental Approaches to Software Engineering (FASE). Lecture Notes in Computer Science 2306, pp.: 174 - 188. Springer-Verlag.

[5] Dörr, H. 1995. *Efficient Graph Rewriting and its implementation.* Lecture Notes in Computer Science, 922. Springer.

[6] 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.

[7] Minas, M. 2002. *Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation* Science of Computer Programming 44, pp.: 157-180.

[8] OMG Home Page: `http://www.omg.org`

[9] Posse, E., de Lara, J., Vangheluwe, H. 2002. *Processing Causal Block Diagrams with Graph Grammars in AToM^3*. In Proceedings of Applied Graph Transformations, AGT'2002. pp.: 23-34. Grenoble.

[10] Python home page: `http://www.python.org`

[11] Vangheluwe, H. 2000. *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. Anchorage, Alaska.