Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3

Juan de Lara[1], **Hans Vangheluwe**[2], **Manuel Alfonseca**[1]

[1] Escuela Politécnica Superior, Ingeniería Informática, Universidad Autónoma de Madrid, 28049 Madrid, Spain
[2] School of Computer Science, McGill University, 3480 University Street, H2A 2A7 Montréal, Québec, Canada

Received: 30 January 2003/Accepted: 13 October 2003
Published online: 1 April 2004 – © Springer-Verlag 2004

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: edit, simulate, transform into another formalism, optimize and generate code. We store all (meta-)models as graphs, and thus, express model manipulations as graph grammars.

We present the design and implementation of these concepts in AToM^3 (A Tool for Multi-formalism, Meta-Modelling). AToM^3 supports modelling of complex systems using different formalisms, all meta-modelled in their own right. Models in different formalisms may be transformed into a single common formalism for further processing. These transformations are specified by graph grammars. Mosterman and Vangheluwe [18] introduced the term multi-paradigm modelling to denote the combination of multiple formalisms, multiple abstraction levels, and meta-modelling. As an example of multi-paradigm modelling we present a meta-model for the Object-Oriented Continuous Simulation Language OOCSMP, in which we combine ideas from UML class diagrams (to express the OOCSMP model structure), Causal Block Diagrams (CBDs), and Statecharts (to specify the methods of the OOCSMP classes). A graph grammar is able to generate OOCSMP code, and then a compiler for this language (C-OOL) generates Java applets for the simulation execution.

Keywords: Meta-modelling – Multi-formalism – Multi-paradigm modelling – Graph grammars – Model transformation – Code generation – Causal block diagrams – Statecharts – AToM^3 – OOCSMP

This is a revised and extended version of a paper presented at the GT-VMT'02 workshop, see [10]

1 Introduction

Complex systems are characterized by components and aspects whose structure as well as behaviour cannot be described in a single formalism (due to their different nature). For example, if we wish to model a temperature and level controlled vessel, the controller can be described with a discrete formalism (such as Petri-Nets or Statecharts [15]) whereas the behaviour of the liquid (which describes the variation in volume and temperature) should be described using a continuous formalism (such as Ordinary Differential Equations or CBDs).

One of the approaches to tackle complex systems is multi-formalism modelling. In this approach the different parts of the system are modelled using different formalisms. In order to analyze a multi-formalism system, it is not enough to look at each component in isolation. One must consider the whole system. For this reason, multi-formalism modelling attempts to convert all components into a common formalism which is closed under composition, so that the whole system can be properly analyzed or simulated.

In order to make the multi-formalism approach possible, we still have to solve the problem of dealing with a plethora of different formalisms. One would like to dedicated tools to model in each one of these formalisms, but the cost of building such tools from scratch is prohibitive. Meta-Modelling is useful to deal with this problem, as it allows the (possibly graphical) modelling of the formalisms themselves. 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's syntax. The advantage of this meta-modelling approach is clear: rather than 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 obvious.

At the very 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 [22]. In Multi-Paradigm Modelling and Simulation we are interested in model manipulations such as:

– Model simulation or animation.
– Model optimization, for example, to reduce its complexity.
– Model transformation into another model (equivalent in behaviour), expressed in a different formalism.
– Generation of (textual) model representations for use by existing simulators or tools. In this paper we will focus on this kind of model transformation.

In this article, we present AToM^3 [3, 8], 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* (ASGs), a generalization of the concept of *Abstract Syntax Trees* used by compilers. The ASG represents – in the form of a graph – the syntactic information of the model built by the user. As a consequence, model manipulation can be expressed as graph grammars.

As an example, we show the generation of a tool to graphically build OOCSMP [2] models. OOCSMP is a (textual) object-oriented continuous simulation language whose development started in Madrid in 1997. In this paper we define a meta-model similar to UML class diagrams [6]. In contrast to UML class diagrams, methods can be described using graphical formalisms used commonly for simulation, such as Statecharts and Causal Block Diagrams. A model described using the previous meta-model is translated into (textual) OOCSMP using a graph grammar. Once the visual model has been translated into OOCSMP, an OOCSMP compiler can translate it into Java or C++ for simulation.

The article is organized as follows: Sect. 2 presents the main ideas behind Multi-Paradigm Modelling. Section 3 gives an overview of the AToM^3 tool. Section 4 presents the meta-model for OOCSMP and for Causal Block Diagrams in detail. Section 5 gives an overview of the graph grammar for code generation. Section 6 presents an example, in which a model of a hybrid system (a temperature and level controlled vessel) is built and OOCSMP code is generated for subsequent simulation. Section 7 presents some related work and finally Sect. 8 states the conclusions and future work.

2 Computer Aided Multi-Paradigm Modelling

Computer Aided Multi-Paradigm Modelling (CAMPaM) [18, 25] is a research area which has the objective to simplify the modelling of complex systems by combining three different directions of research:

– *Meta-Modelling*, which is the process of modelling formalisms. Formalisms are described as models described in meta-formalisms. The latter are nothing but expressive enough formalisms, such as Entity Relationship diagrams (ER) or UML class diagrams. A model of a meta-formalism is called a meta-meta-model; a model of a formalism is called a meta-model. Table 1 depicts the levels considered in our meta-modelling approach. Note that we only consider three levels, although a meta-formalism mf_1 can be 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 usu-

Table 1. Meta-modelling levels

Level	Description	Example
Meta-Meta-Model	Model describes a formalism that will be used to describe other formalisms. Specified with a meta-formalism	Description of Entity-Relationship Diagrams, UML Class Diagrams
Meta-Model	Model describes a simulation formalism. Specified with a meta-formalism	Description of Deterministic Finite Automata, Ordinary Differential Equations (ODE)
Model	Description of an object. Specified with a formalism	$f'(x) = -\sin x, f(0) = 0$ (in the ODE formalism)

ally 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 Deterministic Finite Automata, different transitions leaving a given state must have distinct labels. This cannot be expressed within ER diagrams alone. Expressing constraints is usually done by adding a constraint language to the meta-formalism. Whereas the meta-formalism frequently uses a graphical notation, constraints are usually given in textual form. For this purpose, some systems [14] (including ours) take advantage of the Object Constraint Language OCL [19] used in the UML. As AToM^3 [3] is implemented in the scripting language Python [21], arbitrary Python code can also be used.

Another alternative to using constraints is to express as graph grammar rules, the kind of editing actions the user can perform at each moment in the modelling phase. This approach is called *syntax-directed* [4]. Other kinds of visual editors are called *free-hand* [17] 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 guaranteed by evaluating the constraints defined at the meta-level (and associated with events) when the user is building the model. In AToM^3, free-hand editing can be combined with the syntax-directed approach by building graph grammar rules for editing tasks. See Sect. 7 for some comments about this.

– *Model Abstraction*, concerned with the relationship between models at different levels of abstraction.

– *Multi-Formalism modelling*, concerned with the coupling of and transformation between models described in different formalisms. In Fig. 1, a part of the "formalism space" is depicted in the form of a Formalism Transformation Graph [24]. The different formalisms are shown as nodes in the graph. The solid arrows between them denote a homomorphic relationship "can be mapped onto". The mapping consists of transforming a model in the source formalism into a behaviourally equivalent one in the target formalism. The dotted, vertical thick arrows denote the existence of a simulator for the formalism, which produces simulation traces. This iterative simulation can be seen as a special case of formalism transformation (into the "*traces*" formalism). The vertical dashed line separates continuous (left) and discrete (right) formalisms, whereas the horizontal dashed line below formalism "*DAE non-causal set*" separates causal (upper) and non-causal formalisms. It can be observed how DEVS [26] (Discrete EVent system Specification) can be a suitable target formalism when the purpose of the transformation is simulation, as DEVS can be simulated by parallel, highly efficient simulators based on the HLA arquitecture.

In our approach, we allow the specification of composite systems by coupling heterogeneous components expressed in different formalisms. For the analysis of its properties, the composite system must be assessed by looking at the *whole* multi-formalism system. That is, its components may have to be transformed to a common formalism, which can be found in the FTG. In our approach formalisms are meta-modelled and stored as graphs. Thus, the transformations denoted by the arrows (both for simulation and for formalism transformation) of the FTG can be modelled as graph grammars. In Sect. 6, we present a simple hybrid sys-

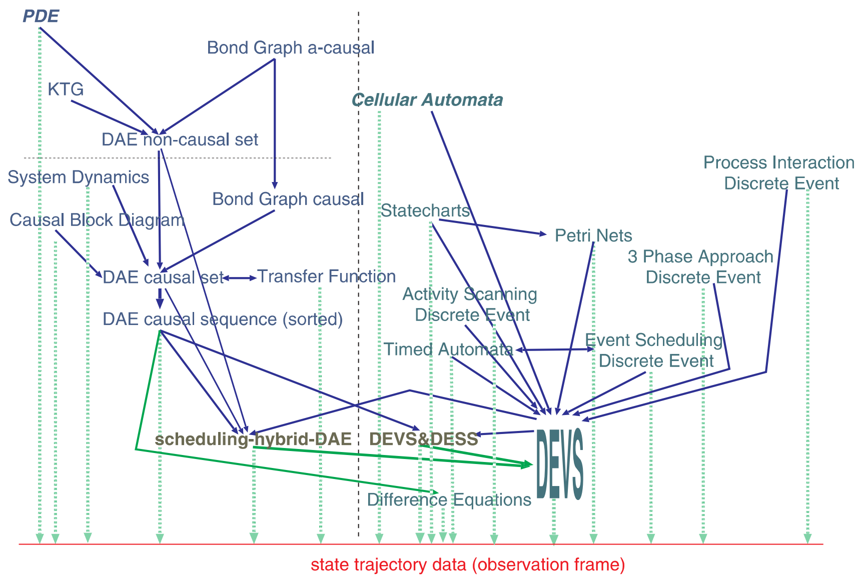


Fig. 1. Formalism Transformation Graph (FTG)

tem specified with Statecharts and Causal Block Diagrams. Both components are transformed into OOCSMP (a causal simulation language whose runtime system can solve Algebraic, Ordinary and Partial Differential Equations) for simulation.

3 AToM^3: An overview

AToM^3 [3, 8] 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) on all platforms for which an interpreter for Python is available: Linux, Windows and MacOS. The main idea of the tool is: "*everything is a model*". During its implementation, the AToM^3 kernel has been bootstrapped from a small initial kernel. Models were defined for bootstrapped parts of it, code was generated and then later incorporated into it. Also, for AToM^3 users, it is possible to modify some of these model-defined components, such as the (meta-)formalisms and the user interface.

AToM^3's architecture is shown in Fig. 2, where models are represented as white boxes, having on their upper-right corner an indication of the formalism they were specified with. In the figure, and for the example in this paper – a graphical representation of OOCSMP models – the meta-meta-model is ER (the *MMF* is also ER, as this meta-formalism was bootstrapped). This meta-formalism is used to describe which are the valid OOCSMP models. The meta-model obtained is thus OOCSMP, the meta-formalism *MF* is ER. Finally, using this OOCSMP meta-model, it is possible to build OOCSMP models such as the one shown in Fig. 8.

The main component of AToM^3 is the Kernel, responsible for loading, saving, creating and manipulating models (at any meta-level, with the *Graph Rewriting Processor* and graph grammar models), as well as for generating code from the (meta-)$^+$models. This code (meta-models and meta-meta-models) can be loaded into AToM^3 as shown in Fig. 2 (*load Formalism* arrows). The first kind of models allows constructing valid models in a certain formalism, the second is used to describe the formalisms themselves.

The ER formalism extended with constraints is available at the meta-meta-level. As stated before, it is perfectly possible to define other meta-formalisms using ER. Constraints can be specified as OCL or Python expressions, and are associated with events (similar to event-programming systems such as Visual Basic). The designer must specify when (pre- or post- and on which event) the condition must be evaluated. Events can be related either to the *abstract syntax* (such as editing an attribute, connecting two entities, etc.) or *purely graphical* (such as dragging, dropping, etc.) If the constraint associated with an event is evaluated to *false*, the event is cancelled, or undone if the constraint was a post-condition. Constraints can be either associated with entities (*local constraints*) or with the whole model (*global constraints*). In AToM^3, we can also define *Actions*, similar to *Constraints* but with side-effects.

When modelling at the (meta-)$^+$level, the entities that may appear in a model must be specified together with their attributes (and constraints and actions as stated before). 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 add the *name* attribute. 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 Fig. 2, labelled 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 (see arrow-

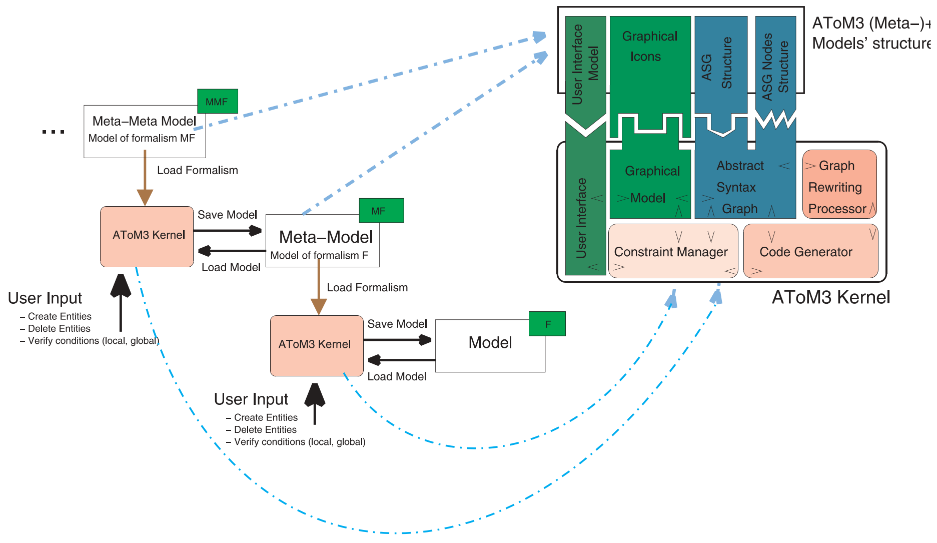


Fig. 2. The architecture of AToM^3

like box labelled as "*User Interface Model*" in the upper-right corner of Fig. 2) . 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. It can be modified by the user to include, for example, buttons to execute graph grammars on the current model. In the example of this paper, we define a graph grammar to generate OOCSMP (textual) code from the OOCSMP meta-model. We have added a button to the user interface to execute this graph grammar, invoke the OOCSMP compiler with the generated code, and execute the resulting simulation applets. AToM^3 generates this user interface model by executing a graph grammar on the meta-model (in the ER formalism) 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. It is envisioned that, in the future, a more complete user interface model will be generated, possibly combining the current "*Buttons*" formalism with a Statechart model of user interaction dynamics.

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 there is no need to distinguish them. A typical example is Statecharts, in which *states* have unnamed ports to connect to other *states. Named* ports are used when we have different meanings for the same types of connections. A typical example of this are the entities 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 signal 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 to be integrated. 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 centre, segments and extremes. For example, for Statecharts, we can choose to represent *States* as ovals with the *name* inside the oval. *Transitions* are arrow-like drawings with the *events*, *conditions* and *actions* besides them. That is, we can specify how some semantic attributes are displayed graphically. Constraints and actions 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 graphical properties (colour, visibility, etc.) That is, in AToM^3, graphical manipulations must be explicitly specified by the user by means of constraints expressed in Python. This is in contrast with other approaches [4] in which constraint languages for graphical layout are used. Python constraints have the drawback of being at a lower abstraction level than a constraint language, but they are usually more efficient.

3.1 Graph transformation in AToM^3

Graph grammars [22] are a generalization of Chomsky grammars, for graphs. They are composed of rules; each having graphs on their left and right hand sides (LHS and RHS). Rules are evaluated against an input graph (called host graph). If a matching is found between the LHS of a rule and a zone in the graph, then the rule can be applied. When a rule is applied, the matching subgraph of the host graph is replaced by the RHS of the rule. Rules can have applicability conditions, as well as actions to be performed when the rule is applied. Some graph rewriting systems have control mechanisms to decide which rule should be checked next. In AToM^3, rules are ordered according to a priority, and are checked from higher to lower priority. After the application of a matching rule, the system again tries to match, starting from the higher priority rule in the list. The graph grammar execution ends when no more applicable rules are found.

Model manipulations can be expressed in AToM^3, either as Python programs or as graph grammar models. The latter has the advantage of being a higher-level, natural, visual, declarative and formal notation. This makes computations become models, easier to specify, understand, and maintain and frees the user of knowing AToM^3 implementation details. The kind of model manipulations we are interested in include model execution, model optimization (for example, reducing its complexity), model transformation into another formalism, and code generation. The latter can be seen as a special case of formalism transformation. As a drawback, the use of graph grammars is constrained by efficiency as 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 and edge types and attributes can greatly reduce the search space.

In Fig. 3, a transformation of a model between two formalisms (F_{source} and F_{dest}) has been depicted. To convert a model from formalism F_{source} to F_{dest} it is necessary to use the meta-models for both F_{source} and F_{dest}, together with the meta-model for graph grammars. The

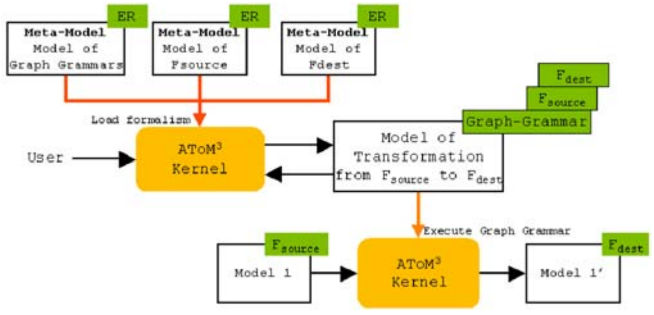
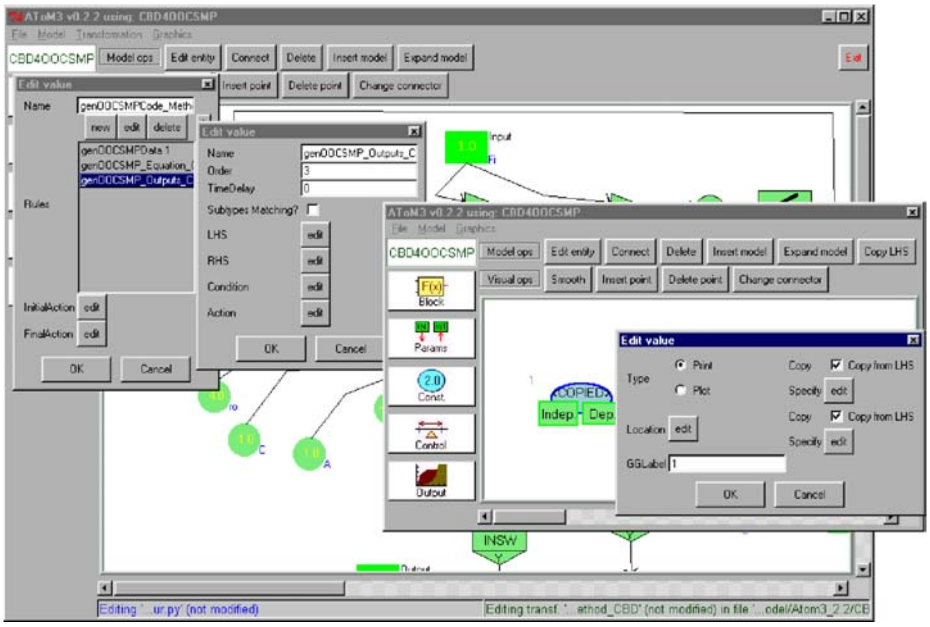

ER
Meta-Model Model of Graph Grammars
ER
Meta-Model Model of Fsource
ER
Meta-Model Model of Fdest
Load formalism
User
AToM3 Kernel
Model of Transformation from Fsource to Fdest
Fdest
Fsource
Graph-Grammar
Execute Graph Grammar
Fsource
Model 1
AToM3 Kernel
Fdest
Model 1'

Fig. 3. Model transformation in AToM^3

graph grammar we are presenting in this paper, which generates OOCSMP textual code from graphical OOCSMP models, is not a model transformation, but a code generation grammar. The OOCSMP models are made of classes in which methods are specified in Statecharts or in the CBD formalism. That is, our source formalism (F_{source} in Fig. 3) is (the graphical) OOCSMP, but we do not need a meta-model for the OOCSMP textual syntax (F_{dest} in Fig. 3). This meta-model would be a description of the *Abstract Syntax Graph* that an OOCSMP compiler builds when parsing an OOCSMP program. Instead, we directly generate OOCSMP textual code from the CBD or the Statecharts model, rather than representing internally the OOCSMP models as *Abstract Syntax Graphs*.

In AToM^3, graph grammars can be graphically edited (as any other model), as shown in Fig. 4. The image shows a moment in the editing of the RHS of a rule. The graph grammar is composed of three rules (see dialog window to the left). Note how in AToM^3 graph grammars can have actions to be executed before and after the graph grammar execution. The next dialog window to the right shows the information about the third rule, named "gen_OOCSMP_Outputs_CBD". In AToM^3 a rule has a *name*, a *priority*, a *time delay*, a flag for *subtype matching*, textual *conditions* and *actions* (expressed in Python) and LHS and RHS models. The *time delay* flag is used if the graph grammar is executed in *animation* mode. Note how this value can be changed by the actions of the rules. Other execution modes for graph grammars are *step-by-step* and *continuous*. The *subtype matching* flag is used as in the matching process we can specify either an exact type matching between the nodes of the LHS and the nodes in the host graph or a "*subtype matching*". In the latter case nodes (or connections) in the LHS and in the host graph do not need to have the same type, but AToM^3 checks at run-time whether the node (or the connection) in the host graph has at least the same set of attributes as the node in the LHS, that is, if the node in the host graph is a structural subtype of the node in the LHS. We do not need to express the subtyping relationship in the meta-models. This relationship is found at run-time. This idea is very useful as one can write very general graph grammars, and reuse them for many formalisms, in unanticipated situations.

The next dialog window to the right of the previous one shows the RHS of the rule being edited. Nodes and edges in LHS and RHS are provided with numbers (the entity we are seeing has been labelled "1") in such a way that if a number appears in both LHS and RHS, the node is not deleted. If it appears in the LHS, but not in the RHS, the node is deleted. Finally, if the number appears in the RHS but not in the LHS, the corresponding node is created. Nodes and connections in the LHS must be provided with the attribute values that will make a match with nodes and connections in the host graph. In AToM^3



Edit value
Name: genOOCSMPCode_Meth
new | edit | delete
Rules: genOOCSMPData 1, genOOCSMP_Equation_(, genOOCSMP_Outputs_C
InitialAction: edit
FinalAction: edit
OK | Cancel

Edit value
Name: genOOCSMP_Outputs_C
Order: 3
TimeDelay: 0
Subtypes Matching? ☐
LHS: edit
RHS: edit
Condition: edit
Action: edit
OK | Cancel

Edit value
Type: (●) Print ( ) Plot
Copy ☑ Copy from LHS
Specify: edit
Copy ☑ Copy from LHS
Specify: edit
Location: edit
GGLabel: 1
OK | Cancel

Editing '...ur.py' (not modified)
Editing transf. '...ethod_CBD' (not modified) in file '...odel/Atom3_2.2/CB

Fig. 4. Editing a Graph Grammar in AToM^3

we can specify that any value of these attributes will make a match, or we can set a specific value. Attributes of nodes and connections in the RHS of a rule, can either maintain the value (checkbox labelled as "Copy from LHS" in the right-most dialog window in the figure), receive a specific value (specified in the different widgets in the dialog window, depending on the attribute's type), or calculate a new one by means of Python code (button labelled as "Specified"). This code can use other node and connection attributes.

4 A graphical representation of OOCSMP models

Our approach for modelling complex systems with AToM^3 is to use the most appropriate formalism for each component in the system, and translate them into a common formalism for simulation. In our case, the common formalism is OOCSMP [2]. This is an Object Oriented extension of the CSMP Continuous Simulation Language [16], developed at the Universidad Autónoma in Madrid. OOCSMP has been extended to handle discrete events, solve Partial Differential Equations, and produce distributed simulations. A compiler (called C-OOL) is able to produce Java applets from the OOCSMP models to perform the simulation. The C-OOL compiler generates a graphical user interface for the simulation that allows the users to experiment, change parameters and answer "*what if..?*" questions. One of the main drawbacks of OOCSMP is the lack of a graphical modelling environment: models are text files which must be coded by hand.

OOCSMP models are made of objects that interact via method invocations. Instructions inside methods are indeed equations, which the compiler sorts appropriately in order to be able to solve them (the language is causal). The main simulation loop is called *DYNAMIC*, and is declared outside any class definition. This 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, which control some of the simulator solver parameters, such as the time step, the interval at which the different outputs get refreshed, etc.

In this section we present a graphical representation for OOCSMP models (built using AToM^3). The idea is to use AToM^3 to describe OOCSMP models in a formalism similar to UML class diagrams, where methods can be described using either CBDs, Statecharts or directly in textual OOCSMP. This allows for the specification of multi-formalism systems in appropriate, graphical and well-known formalisms instead of using a "lower level" textual language such as OOCSMP. The models are then translated into textual OOCSMP code for simulation. This translation is also a model, expressed in the graph grammar formalism. The process is illustrated in Fig. 5. It must be noted that this approach supports meta-model reuse: once we have the meta-model for the structural part of OOCSMP, we can easily add meta-models of formalisms to specify methods with, in a clean, modular way. In the figure, it can be seen that both the Statechart and the CBD meta-models are independent and can be used separately.

The meta-model for the structural part of OOCSMP (the UML class-like notation) is quite simple (see Fig. 6). This meta-model is composed of a single entity called *OOCSMPclass*, which can be related to other *OOCSMPclass* entities by inheritance. These *OOCSMPclasses* have a *name*, a list of *Parameters* (similar to class attributes) and a list of *methods* (specified in CBDs, Statecharts or directly in textual OOCSMP). The main simulation loop (the *DYNAMIC* section) is specified as the only method of a special object called *Main*.

The *Methods* attribute is a list with elements of a composite type *OOCSMPMethod*. Each such element has

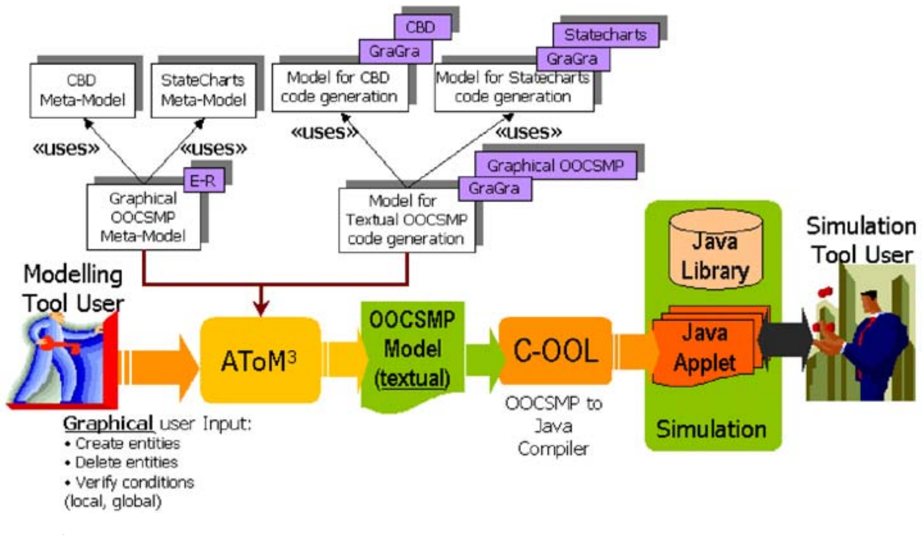


Fig. 5. Using AToM^3 for multi-formalism modelling with OOCSMP (methods can be described with CBDs or statecharts)

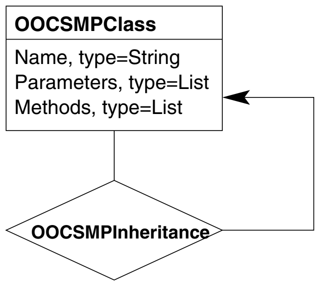


Fig. 6. A meta-model for OOCSMP (structural part)

a name, a list of parameters and a body, which is a model defined either in Statecharts, CBDs or textual OOCSMP. In AToM^3, "types" is just another formalism, and these composite types (to be assigned to attributes) are declared by building a model (as described in [1]). The model for the *OOCSMPMethod* type is shown in Fig. 7. In the figure, it can be seen that an *OOCSMPMethod* (node at the top of the canvas) is a tuple with three components, a *Method_Name* (type String); *Method_Parameters*, a list of *Parameter* entities (a composite type which is a tuple with a name and a value, which can be a scalar, an object or a collection of objects), and the method specification. This last component is a *Union*, which means that it can be specified either in the Statechart or CBD formalisms, or directly in textual OOCSMP syntax. In the latter case, the OOCSMP textual syntax is not checked by AToM^3, this task is performed by the OOCSMP compiler.

In addition, some attributes have been associated with the OOCSMP meta-model as a whole: its *name*, *author*, and *description* on the one hand, and control variable definition on the other. These are variables the user initializes and are used during the simulation to control aspects such as the basic time step (*delta*), the final time (*FINTIM*) and the interval at which variables are printed (*PRdelta*) or plotted (*PLdelta*).

With this information, AToM^3 generates some Python files (see Fig. 2, box labelled as "*AToM^3 (Meta-)$^+$ Models' structure*") which, when loaded in AToM^3, allow the user to graphically build OOCSMP models (see Fig. 8). In the figure, we can see that an OOCSMP class (named *Vessel*) has been defined in the left window. This class has a number of attributes (*T0*, *L0*, *A*, *H*, *C*, all of them of scalar type) and two methods: *Behaviour* and *Controller*. The first is being edited. The dialog window to the right of the previous one (labelled as "Edit value") was automatically generated from the types model shown in Fig. 7 and is being used to edit the properties of method *Behavior*. The method's body is shown in the top-most window, using the CBD meta-model. This method models the variation in temperature and level of the liquid inside a vessel. More details about this example are given in Sect. 6.

We have modified the user interface generated from the OOCSMP meta-model by AToM^3 adding a button to execute a graph grammar for the OOCSMP code generation. Additionally, images have been assigned to the buttons. This can be seen to the left of the background window in Fig. 8. We presented the definition of the Statecharts meta-model with AToM^3 in [9]. The next subsection presents the details of the CBD meta-model.

4.1 Describing the Causal Block Diagrams 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

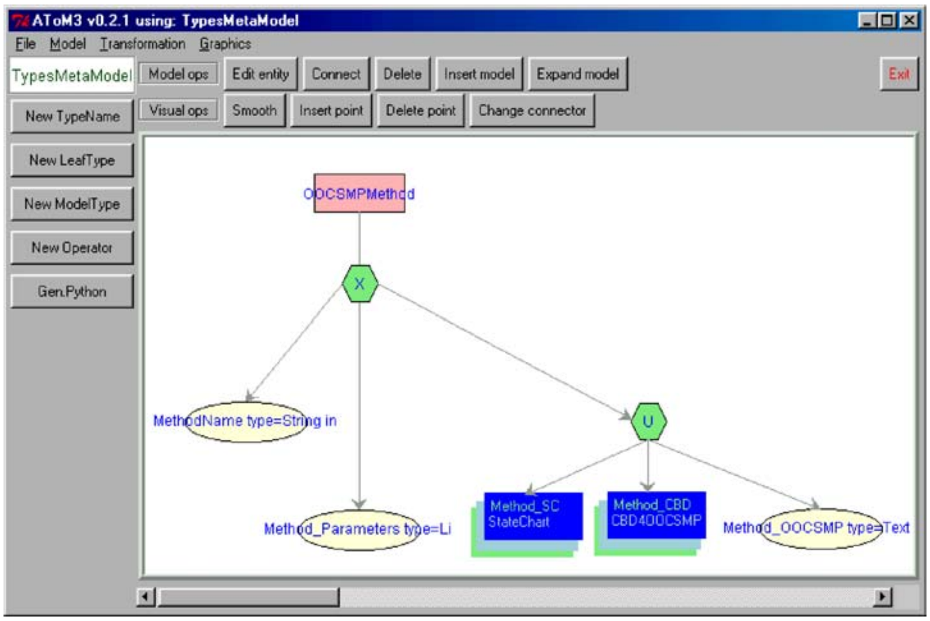


Fig. 7. A model in the "types" formalism which represents an OOCSMP method

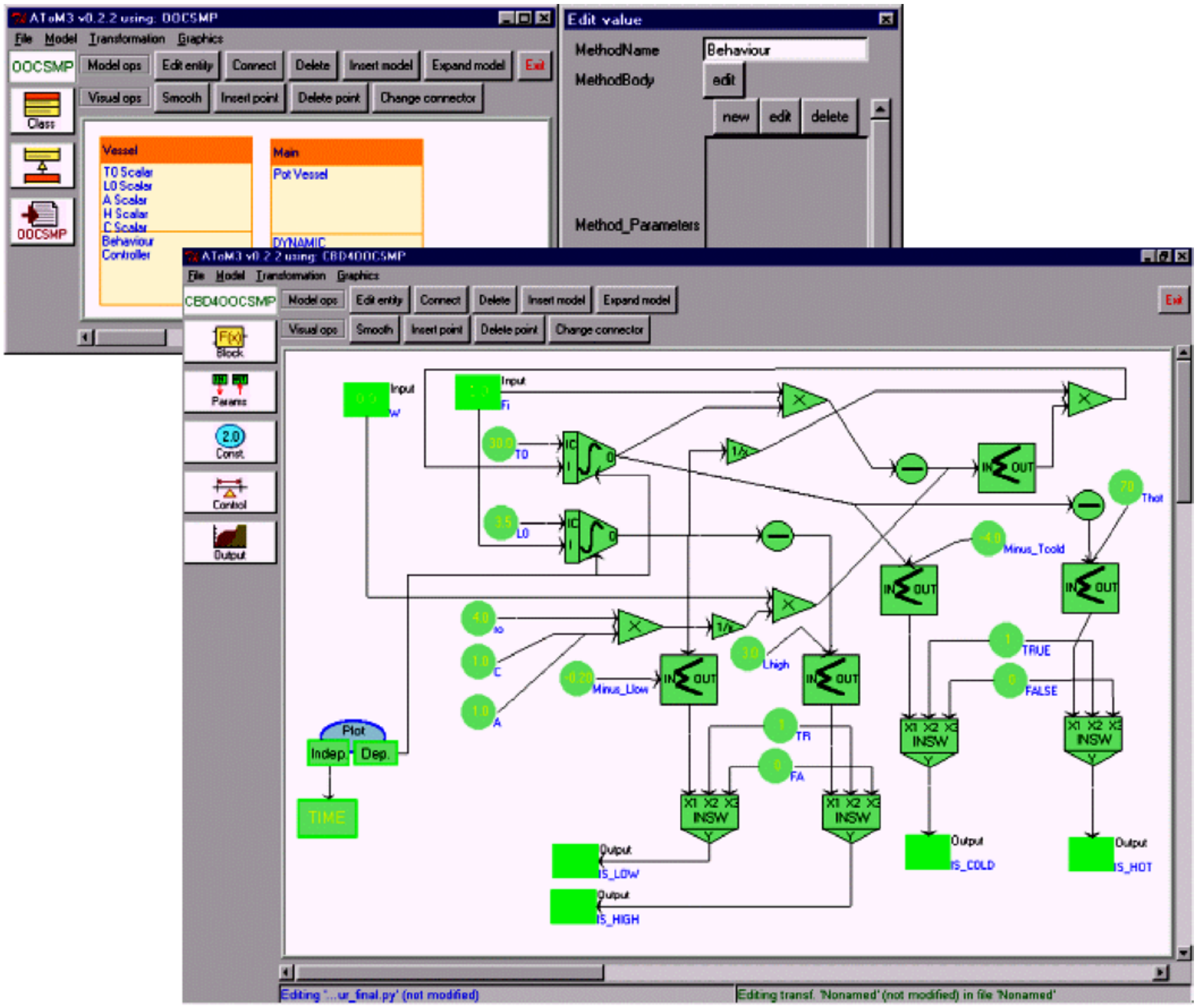


Fig. 8. The generated tool for OOCSMP modelling (defining an OOCSMP method with the CBD formalism)

blocks, these connections transmit *signals* between them. Signals are functions of time. In the meta-model in Fig. 9 we have also included entities to represent constants, control variables (such as *TIME*, the basic time interval, etc.), parameters and outputs. These last elements are 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 *Name* (which is filled automatically with a unique name by AToM^3, but the user can modify it) and 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 functions such as *INTEGRAL* and *DERIVATIVE*. There are 60 blocks in OOCSMP, all of them included in this meta-model. *Block* entities also have a *Value*, which is the result of the application of the block's function to its parameters, six named ports for connecting input blocks, and one named port for output. The input ports are used in some types of *Blocks* (such as *INTEGRAL* and *DERIVATIVE*) to distinguish the inputs. Other blocks (such as the *adder* or the *multiplier*) do not need to distinguish between individual ports.
– *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).
– *Output* entities are used to indicate which *Block* values should be displayed in the simulation (once the model is compiled into OOCSMP). This entity is composed of an attribute named *Type* (to select whether the value should be plotted or printed) and another attribute called *Location* to select in which part of the user interface (automatically generated by *C-OOL*,

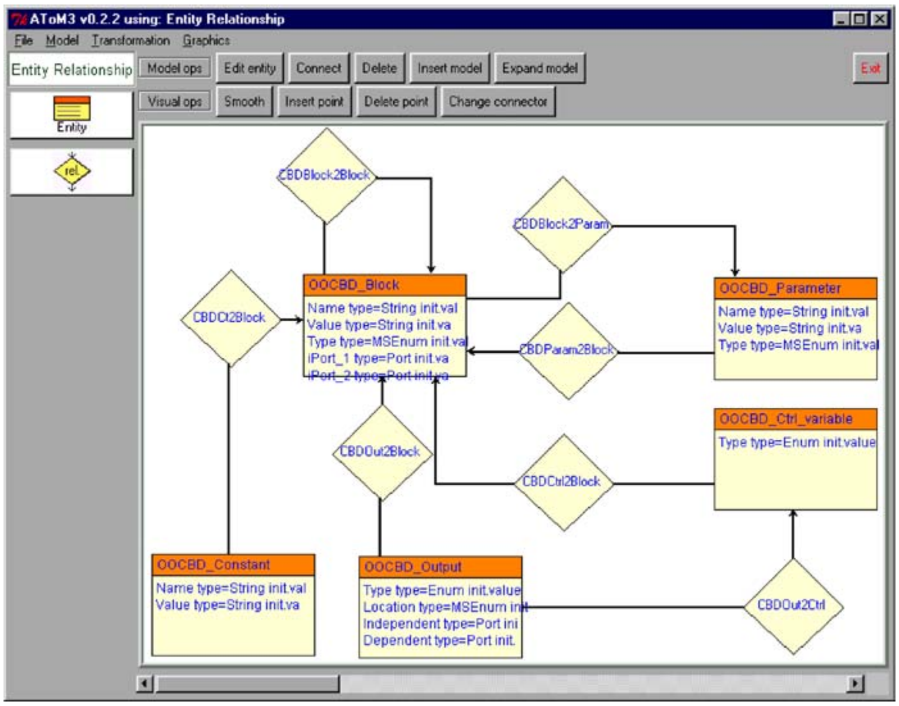

AToM3 v0.2.2 using: Entity Relationship
File Model Transformation Graphics
Entity Relationship
Model ops
Edit entity
Connect
Delete
Insert model
Expand model
Exit
Visual ops
Smooth
Insert point
Delete point
Change connector
Entity
rel.
CBDBlock2Block
CBDBlock2Param
OOCBD_Block
Name type=String init.val
Value type=String init.va
Type type=MSEnum init.va
iPort_1 type=Port init.va
iPort_2 type=Port init.va
OOCBD_Parameter
Name type=String init.val
Value type=String init.va
Type type=MSEnum init.val
CBDCt2Block
CBDParam2Block
CBDOut2Block
OOCBD_Ctrl_variable
Type type=Enum init.value
CBDCtrl2Block
OOCBD_Constant
Name type=String init.val
Value type=String init.va
OOCBD_Output
Type type=Enum init.value
Location type=MSEnum init
Independent type=Port ini
Dependent type=Port init.
CBDOut2Ctrl

Fig. 9. Meta-model of CBDs, expressed in the ER formalism

the OOCSMP compiler) 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 to distinguish between dependent and independent variables while plotting. Dependent variables are calculated by means of expressions possibly containing other dependent and independent variables. Usually, the independent variable is *time*. The independent variable is represented on the X-axis, while dependent variables are represented on the Y-axis.

Figure 10 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

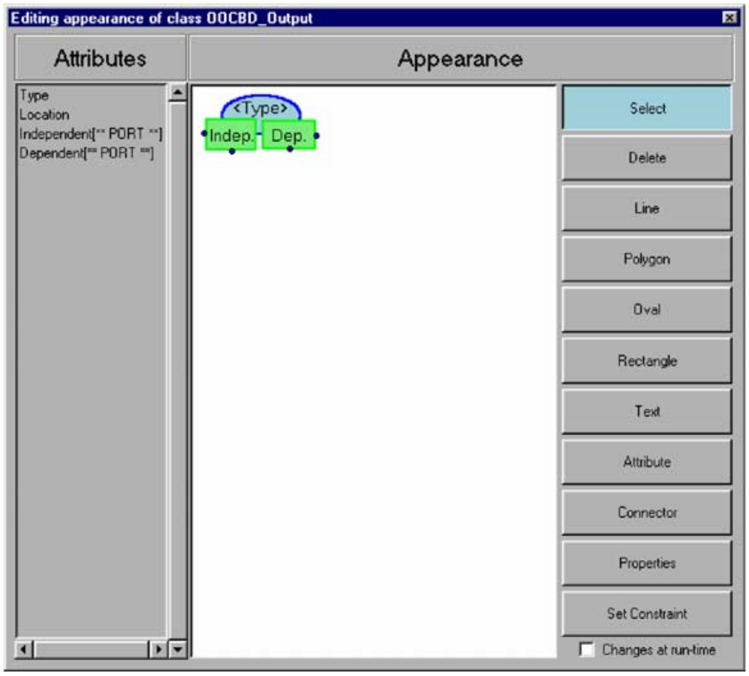


Fig. 10. Defining the graphical appearance of *Output* entities

entity. In this canvas, it is possible to show semantic attributes. In this case the canvas shows the *Type* attribute. Notice that it is also 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 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 labelled *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 those variables 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). For example, *TIME* is often used as the independent variable for plotting, and we frequently place this *control variable* connected to an *Output* entity of type *plot*.
– *Parameter* entities, which are used to input or output values to the outside of this component. These entities have a *Name*, a *Value* and a *Type* that can be either *Input*, *Output* or both.

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, as we want the model to be always correct in this respect. 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 Sect. 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 any number of connections greater than one. 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.

The tool generated from this meta-model description can be seen in the top-most window in Fig. 8. We have modified the default user interface model generated by AToM^3, in such a way that we have only left the buttons to create *Blocks*, *Outputs*, *Parameters*, *Control Variables* and *Constants*. The buttons to create relationships have been eliminated, as relationships are created when connecting entities.

5 Generating OOCSMP code

As was mentioned before, one of the main drawbacks of OOCSMP is the lack of a graphical modelling environment. Models are textual files which must be coded by hand. The work in this paper provides such an environment, replacing the OOCSMP syntax by graphical, well-known simulation formalisms. This has the advantage that one does not have to remember the exact OOCSMP syntax, but only use graphical simulation formalisms. That is, the user can use graphical modelling formalisms, as opposed to coding directly in the "low-level" OOCSMP (low-level, from the point of view of these high-level, graphical formalisms).

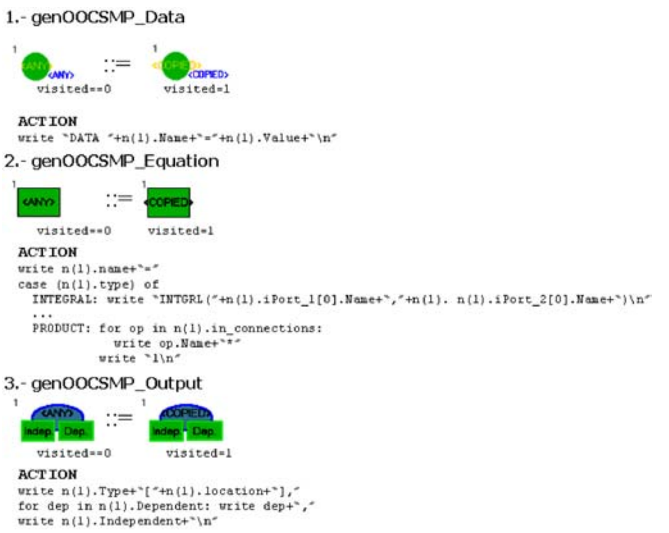
In this section, we show how to produce OOCSMP code for the modelling environment generated in the previous section. This task can be performed with a graph grammar. The initial action of the graph grammar opens a file to store the OOCSMP code and adds an extra attribute (*visited*) to all the nodes of the graph (that is, to all the classes in the model). This attribute controls whether code for that node has been already generated, and is initialized to 0. The initial action also writes in the file the name of the simulation model and of the author.

The only rule of the graph grammar traverses each of the classes declared in the model, generating code for the attributes. Then, for each method, the rule calls another graph grammar, specifically built for the formalism in which this method is expressed. The next subsection explains the graph grammar for code generation from CBD. The called graph grammar generates code for the method, specified either in Statecharts, in CBD or in textual OOCSMP. The advantage of this approach is that graph grammars that generate OOCSMP code from different formalisms are independent and modular.

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 Sect. 4).

Of course, there are more efficient ways to generate code from visual models than by using a graph grammar. For example, coding the algorithm in Python and accessing the AToM^3 API for retrieving the model elements. But graph grammars provide high level control mechanisms which allow the user to perform complex manipulations and graph matching, and the user can specify model manipulations without too much knowledge of the AToM^3 internals.

The next subsection presents the graph grammar to generate OOCSMP code from methods described in the CBD formalism.



```
ACTION
write "DATA "+n(1).Name+"="+n(1).Value+"\n"
```

```
ACTION
write n(1).name+"="
case (n(1).type) of
  INTEGRAL: write "INTGRL("+n(1).iPort_1[0].Name+","+n(1). n(1).iPort_2[0].Name+")\n"
  ...
  PRODUCT: for op in n(1).in_connections:
             write op.Name+"*"
           write "1\n"
```

```
ACTION
write n(1).Type+"["+n(1).location+"],"
for dep in n(1).Dependent: write dep+","
write n(1).Independent+"\n"
```

Fig. 11. Graph Grammar for OOCSMP code generation from methods specified as block diagrams

5.1 Generating OOCSMP code from the methods specified in the CBD formalism

The initial action of this graph grammar adds an extra attribute (*visited*) to all the nodes of the graph except *Control Variables* (that is, to all *Constant*, *Block* and *Output* nodes). This attribute controls whether code for that node has already been generated, and is initialized to 0. The graph grammar is composed of three rules, none of which changes the matching subgraph. It is shown in Fig. 11. We use the notation "*n(x)*" in the "*action*" section to refer to the node in the host graph which matches the LHS node whose label is "x".

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. 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. 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 (plotted or printed). Note how, in the "*action*" sections, we are accessing the connected elements of the nodes (through ports), as these may have an arbitrary number of connections. Another approach is to use similar ideas with amalgamated and parallel graph transformation [23], which allow dealing with variable context by combining and syncronizing productions.

6 Example: A temperature and level controlled vessel

As an example of how to apply the concepts and tools presented above, we introduce a toy model to illustrate the usefulness of the approach. In the example, we consider a temperature and level controlled liquid in a vessel. This is a modification of the system described in [5], where structural change is the main issue. On the one hand, the liquid can be heated or cooled; on the other hand, liquid can be added or removed (we do not consider phase changes for this simple example). The liquid's temperature T and level L are governed by the following Ordinary Differential Equation model:

$$\frac{dT}{dt} = \frac{1}{L}\left[\frac{W}{c\rho A} - \phi T\right] \quad (1)$$

$$\frac{dL}{dt} = \phi \qquad \text{if } 0 < \text{L} < \text{H else } 0 \quad (2)$$

$$is\_low = (L < L_{low}) \quad (3)$$

$$is\_full = (L > L_{high}) \quad (4)$$

$$is\_cold = (T < T_{cold}) \quad (5)$$

$$is\_hot = (T > T_{hot}) \quad (6)$$

The inputs to the model are ϕ, the flow rate, and W, the rate at which heat is added or removed. The model's parameters are A, the cross-section surface of the vessel, H, its height, c, the specific heat of the liquid and ρ, the liquid density. Equations (3)–(6) set threshold output sensors, which are the outputs of this model and are sent to the discrete-event controller. Equations (1)–(6) have been included in method *Behaviour* in class *Vessel* (see Fig. 8) using CBDs. Note how, in the model, the *is_low*, *is_full*, *is_cold* and *is_hot* flags appear as green boxes (parameters), labelled as "Output" at the bottom of the model. Their value is calculated with the OOCSMP block *INSW*, which returns the second argument if the first argument is less than zero, else it returns the third argument. The inputs of the model appear labelled as *Fi* and *W*, to the top of the window. Note also how we are plotting the value of L and T with respect to time.

The other part of the system is a controller for the temperature and the level, which tries to maintain both quantities between certain limits. This is implemented as a Statechart (shown in Fig. 12) and is embedded in method *Controller* in the *Vessel* class. The idea is to have an orthogonal component for the temperature and another one for the level. Each component has three states, corresponding to the situation in which the quantity to control (level or temperature) is high, medium or low. The possible events (that is, the model's inputs) which make the system change its state are *is_low*, *is_high*, *is_cold* and *is_hot*. As associated actions to these events, ϕ and W are modified. For example, if the system is in state *cold* in orthogonal component *temperature* and receives the event *IS_COLD*, it increases W by some fixed amount DW and remains in the same state (responding to subsequent *IS_COLD* events in the same way) until the event *IS_COLD* is no longer sent by the *Behaviour* model. ϕ and W are the outputs of this model, which are fed to method *Behaviour*. Finally, the main simulation loop is modelled as textual OOCSMP code and contains

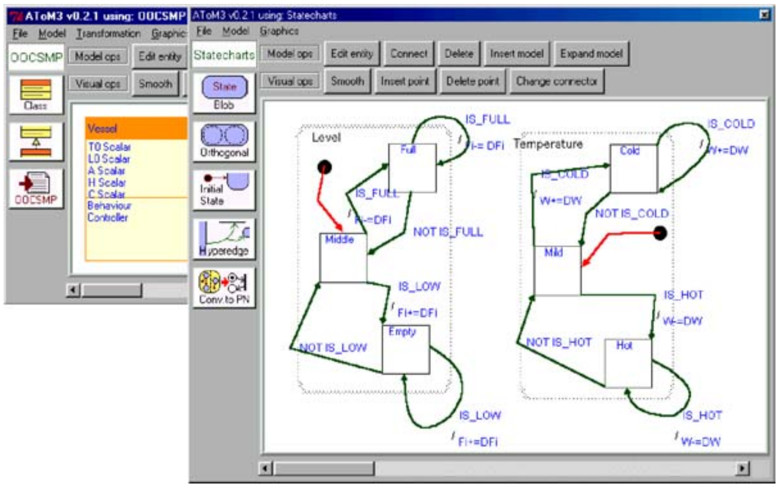
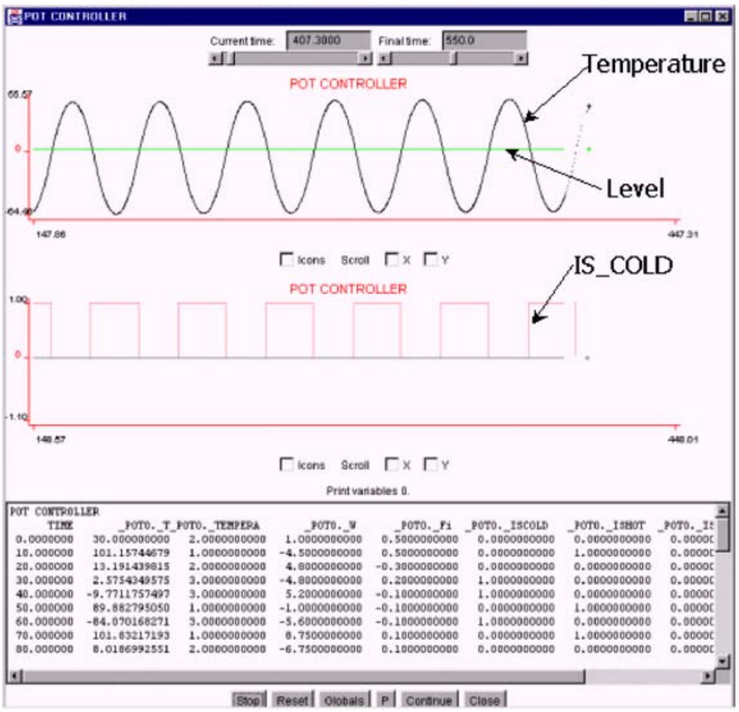


Fig. 12. Defining the controller for the Vessel in the Statecharts formalism



```
POT CONTROLLER
      TIME      _POTO._T_POTO._TEMPERA       _POTO._W      _POTO._Fi  _POTO._ISCOLD   _POTO._ISHOT  _POTO._IS
 0.0000000   30.000000000   2.0000000000   1.0000000000   0.5000000000   0.0000000000   0.0000000000   0.0000C
 10.000000   101.15744679   1.0000000000  -4.5000000000   0.5000000000   0.0000000000   1.0000000000   0.0000C
 20.000000   13.191439815   2.0000000000   4.8000000000  -0.3000000000   0.0000000000   0.0000000000   0.0000C
 30.000000   2.5754349575   3.0000000000  -4.8000000000   0.2000000000   1.0000000000   0.0000000000   0.0000C
 40.000000  -9.7711757497   3.0000000000   5.2000000000  -0.1000000000   1.0000000000   0.0000000000   0.0000C
 50.000000   89.882795050   1.0000000000  -1.0000000000  -0.1000000000   0.0000000000   1.0000000000   0.0000C
 60.000000  -84.070168271   3.0000000000  -5.6000000000  -0.1000000000   1.0000000000   0.0000000000   0.0000C
 70.000000   101.83217193   1.0000000000   8.7500000000   0.1000000000   0.0000000000   1.0000000000   0.0000C
 80.000000   8.0186992551   2.0000000000  -6.7500000000   0.1000000000   0.0000000000   0.0000000000   0.0000C
```

Fig. 13. A moment in the simulation of the vessel model

the appropriate invocation of methods *Behaviour* and *Controller*.

This model is transformed into OOCSMP for simulation using the graph grammar defined in Sect. 5. Once the model has been compiled into Java with C-OOL, we can execute the simulation. A moment in the simulation is shown in Fig. 13. In the upper plot, the variation of temperature and level is shown. After some transitory effects at the beginning of the simulation (not shown), the temperature reaches an oscillatory equilibrium. In the middle plot the values of *IS_COLD*, *IS_HOT*, *IS_LOW* and *IS_HIGH* are shown. It can be observed how *IS_COLD* is switching from 0 to 1 periodically to maintain the temperature between the limits. The listing at the bottom shows some of the variable values. This simulation applet is available at `http://www.ii.uam.es/~jlara/investigacion/ecomm/hybrid.html` for the reader interested in experimenting with it.

7 Related work

There are some similar tools in the graph grammars community, such as GenGed [4], which builds *syntax directed* or *free-hand* visual modelling environments. Their ideas are similar to ours, but we 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. Sometimes, this can be more difficult than using constraints, but our approach offers more efficiency as constraint solvers tend to be slow. Other systems also make a clear distinction between abstract and concrete (graphical information) syntax, with a flexible mapping between both models. The current version of AToM^3 only supports a one-to-one mapping from entities into icons and relationships into arrows.

DiaGen [17] is a tool based on *hypergraph* grammars that may combine characteristics of *free-hand* and *syntax-directed* editors. The user inputs a textual specification of the visual language and obtains a set of Java classes which are complemented by a Java library to obtain the visual environment. In AToM^3, the specification of the visual language (the meta-model) is done graphically, and the generated files are loaded again into AToM^3. There is no structural difference between the generated editors (which could be used to generate other ones!), and the editor which generated them. In fact, one of the main differences of the approach taken in AToM^3 with other similar tools, 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 modify it.

In the simulation community, with respect to complex systems modelling, the approach of [20] is similar to ours. They have implemented several editors for continuous (sequential function charts) and discrete formalisms (Statecharts) using the meta-modelling tool DoME [11]. The user builds composite models with these editors. Models are subsequently translated into the object oriented simulation language MODELICA [12]. In DoME, model manipulations must be expressed either in the Lisp-like language Alter or in Smalltalk; in our approach they can be visually specified by means of graph grammars (combined with Python if desired).

A different approach to the modelling of complex systems is Paul Fishwick's Multi-Modelling [13]. His approach deals with multi-formalism (a *multi-model* can be composed of components described in different formalisms) and multiple abstraction levels. To synchronize the system at its highest level, a coordinator is used to direct the events to the appropriate models (co-simulation). The approach of the Ptolomey [7] environment is another example of the power of this approach. It is noted that its power is partly due to the fact that it is an integrated environment, which does not need to interface with external simulations.

Our approach (*Multi-Paradigm Modelling*) proposes *Meta-Modelling* as a method to obtain a tool to model in each formalism, and translates models between formalisms for the purpose of simulation and analysis. This previous step (translation) to simulation makes the simulation process cleaner and permits analysis of properties of the multi-formalism model with tools available in the base formalism. In addition, when a model is translated into a formalism, there are possibilities to apply optimizing transformations. If the translation process goes through multiple formalisms, then one can apply optimizing transformation in each formalism. For example, if all the components of a multi-formalism system are translated, say, to Petri-Nets, then one can use complexity reduction transformations and later reachability analysis to verify that certain system states are reachable. In contrast, in the *Multi-Modelling* approach, one cannot apply these analysis tools, as each component has been described in a different formalism and no transformation to a common formalism is performed. As stated in the introduction, it is not sufficient to look at properties of the subcomponents in isolation, one should look at the properties of the system *as a whole*.

8 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 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 OOCSMP. For that purpose, we have designed a meta-model similar to UML class diagrams, in which methods can be described using Statecharts, CBDs or OOCSMP textual code. Thus, the user does not have to know the OOCSMP syntax, but may use some well-known simulation formalisms. These are then translated into OOCSMP syntax using a graph grammar for subsequent simulation.

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 transformation 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:

- 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 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 (using the Formalism Transformation Graph, see [24]).
– 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.
– Extend the user interface model (which is generated by AToM^3 for meta-models) with Statecharts. This will allow the user to control more complex behaviours of the generated tool.
– Provide the tool with automatic layout algorithms for the graphical representation of the models. These can be specially useful after applying a graph grammar which modifies the model's structure.

The Meta Object Facility (MOF) [19] is an OMG standard for a meta-metamodel. Using this MOF Model, one could define different metamodels (for example for UML). Using XMI it is possible to automatically obtain DTDs and XML documents for the models. It is worth exploring either the possibility of meta-modelling the MOF Model or to change the current AToM^3 "hard-wired" primitives for meta-generation to make it "MOF compatible". This would require a completely separate model for the graphical representation of models as well, together with a much more flexible mapping from abstract to concrete syntax (such as in GenGed).

With respect to AToM^3 as a front end for OOCSMP, we would like to improve the graph grammar for code generation from CBD models, to distinguish expressions whose value is not going to change during the simulation. These values may be calculated at the beginning of the simulation (in a section called *INITIAL*) rather than at each time step. We are also working to extend the number of formalisms available to specify OOCSMP models, in particular we are working on meta-modelling Forrester System Dynamics and their translation into OOCSMP.

Acknowledgements. The authors would like to thank three anonymous referees for their useful comments. Juan de Lara and Manuel Alfonseca gratefully acknowledge partial sponsorship for this work by the Spanish Ministry of Science and Technology (MCYT), project number TIC2002-01948. Hans Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant.

References

1. Aho AV, Sethi R, Ullman JD (1986) Compilers, principles, techniques and tools. Chapter 6, Type Checking. Addison-Wesley
2. Alfonseca M, Pulido E, Orosco R, de Lara J (1997) OOCSMP: An Object-Oriented Simulation Language. In: 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`
3. AToM^3 home page: `http://atom3.cs.mcgill.ca`
4. Bardohl R, Ermel C, Weinhold I (2002) AGG and GenGED: Graph Transformation-Based Specification and Analysis Techniques for Visual Languages. In: Proc. GraBaTs 2002. Electronic Notes in Theoretical Computer Science, vol 72(2)
5. Barros FJ, Zeigler BP, Fishwick PA (1998) Multimodels and dynamic structure models: an integration of DSDE/DEVS and OOPM. In: Proceedings of the 1998 Winter Simulation Conference, pp 413–419
6. Booch G, Rumbaugh J, Jacobson I (1999) The Unified Modeling Language User Guide. Addison Wesley
7. Davis II J, Hylands C, Kienhuis B, Lee EA, Liu J, Liu X, Muliadi L, Neuendorffer S, Tsay J, Vogel B, Xiong Y (2001) Heterogeneous Concurrent Modeling and Design in Java. Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley. See also: `http://ptolomey.eecs.berkeley.edu/publications`
8. 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 ETAPS'02, Fundamental Approaches to Software Engineering (FASE). Lecture Notes in Computer Science, vol 2306. Springer-Verlag, pp 174–188
9. de Lara J, Vangheluwe H (2002) Computer Aided Multi-Paradigm Modelling to process Petri-Nets and Statecharts. In: 1st International Conference on Graph Transformations, ICGT'2002 (Barcelona). Lecture Notes in Computer Science, vol 2505, pp 239–253
10. de Lara J, Vangheluwe H, Alfonseca M (2002) Using Meta-Modelling and Graph Grammars to create Modelling Environments. In: Graph Transformations and Visual Modelling Techniques (GT-VMT) Workshop, Barcelona. Electronic Notes in Theoretical Computer Science, vol 72(3)
11. DOME guide (2000) `http://www.htc.honeywell.com/dome/`, Honeywell Technology Center. Honeywell, version 5.3
12. Elmqvist H, Mattson SE (1997) An Introduction to the Physical Modeling Language Modelica. In: Proceedings 9th European Simulation Sympossium ESS97, SCS Int., Erlangen, pp 110–114. See also `http://www.modelica.org`
13. Fishwick P, Zeigler BP (1992) A Multimodel Methodology for Qualitative Model Engineering. ACM Transactions on Modelling and Computer Simulation 1(2):52–81
14. Gray J, Bapty T, Neema S (2000) Aspectifying Constraints in Model-Integrated Computing. In: OOPSLA 2000: Workshop on Advanced Separation of Concerns, Minneapolis, MN, October, 2000
15. Harel D (1987) Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8:231–274
16. IBM Corp. (1972) Continuous System Modelling Program III (CSMP III) and Graphic Feature (CSMP III Graphic Feature) General Information Manual. IBM Canada, Ontario, GH19-7000
17. Minas M (2002) Specifying Graph-like diagrams with DIAGEN. Science of Computer Programming 44:157–180
18. Mosterman P, Vangheluwe H (2002) Computer Automated Multi-Paradigm Modeling. ACM Transactions on Modeling and Computer Simulation 12(4):1–7. Special Issue Guest Editorial
19. OMG Home Page: `http://www.omg.org`
20. Pereira Remelhe M, Engel S, Otter M, Derarade A, Mosterman P (2002) An Environment for Integrated Modelling of Systems with Complex Continuous and Discrete Dynamics. In: Lecture Notes in Control and Information Systems, vol 279, pp: 83–105
21. Python home page: `http://www.python.org`
22. Rozenberg G (ed) (1999) Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1. World Scientific
23. Taentzer G (1996) Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems. PhD Dissertation, Shaker Verlag

24. Vangheluwe H (2000) DEVS as a common denominator for multi-formalism hybrid systems modelling. In: Varga A (ed) IEEE International Symposium on Computer-Aided Control System Design. IEEE Computer Society Press, Anchorage, Alaska, pp 129–134
25. Vangheluwe H, de Lara J, Mosterman P (2002) An Introduction to Multi-Paradigm Modelling and Simulation. In: Proceedings of AI, Simulation and Planning – AIS'2002. Lisbon. SCS International, pp: 9–20
26. Zeigler BP, Praehofer H, Kim TG (2000) Theory of modelling and simulation: Integrating discrete event and continuous complex dynamic systems, second ed. Academic Press

Juan de Lara is an an assistant professor at the Computer Science Department of the Universidad Autónoma in Madrid, where he teaches Software Engineering. He holds a PhD degree in Computer Science, and works in areas such as Web based Simulation, Agent based Simulation and Multi-Paradigm Modelling. In the latter area he worked as a postdoc at the MSDL lab headed by prof. Hans Vangheluwe.
His e-mail address is `Juan.Lara@ii.uam.es`, and his web page is `www.ii.uam.es/∼jlara`.

Hans Vangheluwe is an assistant professor in the School of Computer Science at McGill University, Montréal, Canada where he teaches Modelling and Simulation, as well as Software Design. He also heads the Modelling, Simulation and Design Lab (`http://msdl.cs.mcgill.ca`). Some of his model compiler work has led to the WEST++ tool, which was commercialised for use in the design and optimization of Waste Water Treatment Plants. He was the co-founder and coordinator of the European Union's ESPRIT Basic Research Working Group 8467 "Simulation in Europe", and a founding member of the Modelica Design Team.
His e-mail address is `hv@cs.mcgill.ca`, and his web page is `www.cs.mcgill.ca/∼hv`.

Manuel Alfonseca is director of the Higher Polytechnical School in the Universidad Autónoma of Madrid. From 1972 to 1994 he was Senior Technical Staff Member at the IBM Madrid Scientific Center. He works on simulation, complex systems and theoretical computer science.
His e-mail address is `Manuel.Alfonseca@ii.uam.es`, and his web page is `www.ii.uam.es/∼alfonsec`.