Processing Causal Block Diagrams with Graph Grammars in AToM^3

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

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

Abstract. AToM^3 is a tool which supports *multi-formalism* modelling and *meta-modelling* to facilitate computer assisted analysis and design of complex systems. To enable the automatic generation of modelling tools, the formalisms themselves are modelled at a meta-level within an appropriate meta-formalism. The generated tools are able to process (create, edit, simulate,...) models expressed in the corresponding formalism. AToM^3 relies on graph grammars and graph rewriting techniques to perform the transformations between formalisms as well as for other tasks, such as code generation, model optimization and simulator specification. As a case study, we describe the syntax and operational semantics of Causal Block Diagrams (CBD). The animation of such operational semantics results in the actual simulation.

Keywords: Modelling & Simulation, Meta-Modelling, Multi-Formalism Modelling, Graph Grammars, Operational Semantics.

1 Introduction

AToM^3 is a visual Meta-Modelling tool developed by the authors, which supports modelling of complex systems. Complex systems are characterized by components and aspects which, in addition to being numerous, have structure and behaviour which cannot be appropriately described in a single formalism. Examples of commonly used modelling formalisms are Differential-Algebraic Equations (DAE), Causal Block Diagrams, Petri Nets, Entity-Relationship diagrams (ERD), and State Charts.

From the meta-specification of a modelling formalism, AToM^3 is able to produce customized tools to process models specified in the described formalism. Both syntax and semantics of a formalism are modelled. Some of the model manipulations in which we are interested include transformations to other formalisms, simulations, optimizations and (textual) code generation for other tools.

Causal Block Diagrams (CBD) are a general formalism used for modelling of causal, continuous-time systems. The simulation of such systems on digital computers requires a discrete-time approximation. There are several approaches to this simulation problem. One interesting solution is to describe CBD syntax in an appropriate CBD meta-model and to provide a specification of the operational semantics of such diagrams using graph grammars. The animation of such operational semantics will result in the actual simulation. We can thus regard the graph grammar as an *executable specification*. This approach is desirable for its generality, since it can be applied to a wide class of formalisms besides CBD. There is, however, a tradeoff made between generality and efficiency. As a general rule, customized, hand-coded, formalism-specific simulation algorithms are more efficient. The approach of relying on graph grammars is expensive due to the nature of the graph matching algorithm. However, there are other motivations, both theoretical and practical:

- Explicitly defining the operational semantics of any formalism should be considered as part of the design of the actual simulator, providing a specification, from which a more efficient implementation could be built.
- The specification also provides a framework (a reference implementation) for verifying and testing different implementations.
- It provides a portable simulator, since it is more abstract than a hand-coded implementation.
- It allows for reasoning about the described systems. For example, it allows for the definition of general algorithms for bisimulation.

The rest of the paper is organized as follows. Section 2 describes the motivations for meta-modelling. Section 3 relates graph-grammars to meta-modelling. Section 4 gives a brief description of AToM^3's architecture. Section 5 describes the specification (meta-model) of the CBD formalism. Section 6 provides the definition of CBD's semantics in terms of graph grammars.

2 Meta-Modelling

One of the characteristics of complex systems is the diversity of their components. Consequently, it is often desirable to model the different components using different modelling formalisms. This is certainly the case when inter-disciplinary teams collaborate on the development of a single system. Flexibility is also required as different teams may prefer slight variations of a particular formalism. A proven method to achieve the required flexibility for a modelling language that supports many formalisms and modelling paradigms is to model the modelling language itself [4][10]. Such a model of the modelling language is called a meta-model. It describes the possible structures which can be expressed in the language. A meta-model can easily be tailored to specific needs of particular domains. This requires the meta-model modelling formalism to be rich enough to support the constructs needed to define a modelling language. Taking the methodology one step further, the meta-modelling formalism itself may be modelled by means of a meta-meta-model. This meta-meta-model specification captures the basic elements needed to design a formalism. Table 1 depicts the levels considered in our meta-modelling approach.

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.

Formalisms such as ERD are often used for meta-modelling. To be able to fully specify modelling formalisms, the meta-level formalism may have to be extended with the ability to express constraints (limiting the number of meaningful models). For example, when modelling a Determinsitic Finite Automaton, different transitions leaving a given state must have different labels. This cannot be expressed within the ERD formalism alone. Expressing constraints is most elegantly done by adding a constraint language to the meta-modelling formalism. Whereas the meta-modelling formalism frequently uses a graphical notation, constraints are concisely expressed in textual form. For this purpose, some systems [6], including AToM^3 use the Object Constraint Language OCL [8] used in the UML.

Fig. 1 depicts the structure we propose for a meta-modelling environment. AToM^3 was initialized using a hand-coded ERD meta-meta-model. As the ERD formalism can be described as an ERD model, the environment was subsequently bootstrapped. Meta-formalisms are described by meta-meta-models. Although it is possible to describe a meta-formalism mf_1 using another meta-formalism mf_2 we consider both as meta-formalisms as no more capabilities are added by going to higher meta-levels.

3 Graph grammars and Meta-modelling

Graph-grammars play an important role in our approach to the modelling of complex systems. We represent models as Abstract Syntax Graphs (as a logical generalisation of Abstract Syntax Trees), and therefore model processing as graph grammars. Some of the manipulations we are interested in are:

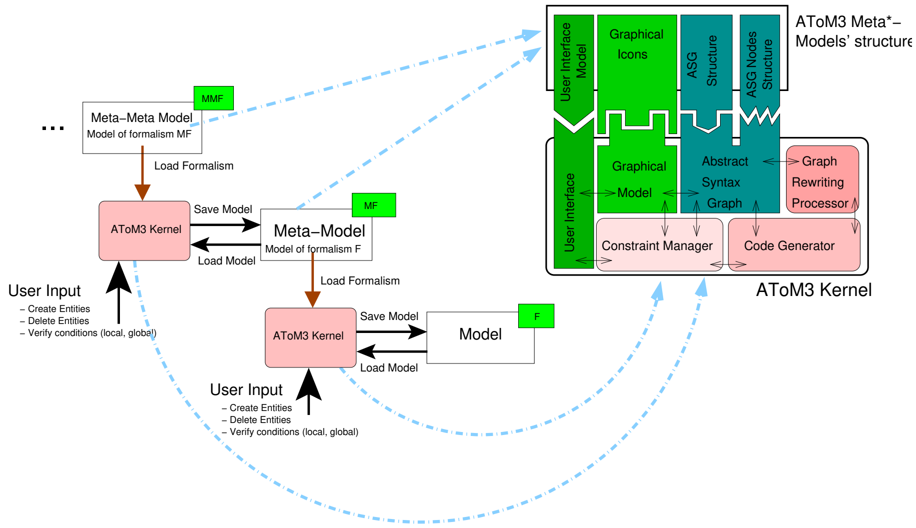

Fig. 1. Proposed working scheme for a meta-modelling environment.

- Formalism transformation: Given a model in a certain formalism, these transformations convert it into a model, but expressed in another formalism. For Modelling and Simulation, possible transformations are given in a Formalism Transformation Graph [11].
- Model optimization: These transformations do not change the formalism in which the model is expressed. Their application results in a reduction of the model complexity.
- Code Generation: These transformations produce a textual representation of the model (subject to syntactic constraints).
- Simulator specification: These graph grammars specify the operational semantics of the model. We will present an example of this kind of graph grammar in section 5.

All these tasks depend on the formalisms of interest. However, since models determined by some meta-model are graphs (subject to the constraints given by the meta-model), these tasks can be performed by a generic graph-transformation algorithm. Therefore it makes sense to combine meta-modelling and graph-grammars in a unifying framework. Meta-models determine the classes of graphs that are allowed on the LHS and RHS of a graph-grammar rule. Furthermore, the rules themselves, and the grammars, can be viewed as models in the graph-grammar formalism, which itself can be described in a meta-model.

There are tools for specifying graph-grammars and tools for meta-modelling, but to our knowledge no tool combines them in a unified framework. AToM^3 was conceived to fill this gap.

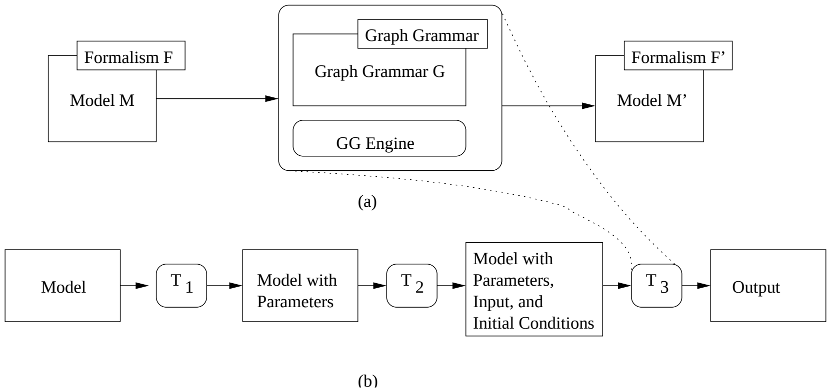

Fig. 2. Graph-grammars in Modelling and Simulation.

We would like to emphasize the role of graph-grammars in Modelling and Simulation. As mentioned before, graph transformations can be regarded as models, which can process models of other formalisms (Fig. 2 (a)). This basic paradigm can be applied to the general process of simulation. In order to simulate a model, one must first provide values to the model's parameters, and feed these, with the actual input, to the simulator ([12]). Each of these processes can be specified by graph grammars (Fig. 2 (b)). In transformations T_1 and T_2, the given model is enriched with additional structure (parameters and input). Transformation T_3 is the actual simulator, which can also be specified as a graph grammar, based on the operational semantics of the model's formalism. Input as well as output, can themselves be regarded as models in a formalism of traces (time-segments) of the values of interest.

4 AToM^3

AToM^3 is a Meta-Modelling tool written in Python [9]. Its main component is the Kernel, which is responsible for loading, saving, creating and manipulating models (at any meta-level), as well as for generating code for customized tools. Both meta-models and meta-meta-models can be loaded when AToM^3 is invoked (see Figure 1). The first kind of models allow construction of valid models in a certain formalism, the latter are used to describe the formalisms themselves. In AToM^3 all models, irrespective of meta-level, have the same internal structure (a graph).

The ERD formalism extended with constraints is available at the meta-meta-level. 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-meta-level, the entities which may appear in a model must be specified together with their attributes. AToM^3 supports 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. In this way a meta-formalism, such as the ERD can be used to describe other meta-formalisms, such as the UML class diagrams. Both types of attributes, *regular* and *generative* may contain data or code for pre- and post-conditions.

The meta-meta-information is used by the *Kernel* to generate some Python files (see upper-right corner of Fig. 1), which, when loaded by the *Kernel*, allows the processing of models in the defined formalism. These files include a model of the user interface presented when the formalism is loaded. This model follows the rules of the *"Buttons"* formalism, and by default contains a button to create each object found in the meta-model. For the case of the Petri-Nets formalism ([7]), it would contain buttons to create *Places*, *Transitions*, and the connections between them. This model can be modified using AToM^3 to for example add buttons to execute graph grammars on the current model or delete unwanted buttons. When a formalism is loaded, the *Kernel* interprets this user interface model, to create and place the actual widgets and associate them with the appropriate actions.

The functionalities of the generated tools include creating models under the rules of the specified formalism, verifying that these models are valid, loading, saving, and producing a Postscript file with its graphical representation. Further model manipulations can be obtained by defining appropriate graph grammars.

For the implementation of the Graph Rewriting Processor, we have used an improvement of the algorithm given in [5], in which we allow non-connected graphs in LHSs. It is also possible to define a sequence of graph grammars that have to be applied to the model. This is useful, for example to couple grammars to convert a model into another formalism, and then apply an optimizing grammar. For clarity and efficiency reasons graph grammars are often divided in independent parts. In our tool, rules are ordered based on a user-assigned priority, and the rewriting system iteratively applies matching rules in the grammar to the graph, until no more rules are applicable.

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

As in grammars for formalism transformations we have a mixing of entities belonging to different formalisms, it must be possible to open several meta-models at the same time. Obviously, the constraints of the individual formalism meta-models are meaningless when entities in different formalisms are present in a single model. Such a model may come to exist during the intermediate stages of graph grammar evaluation when transforming a model from one formalism into another. It is thus necessary to disable evaluation of constraints during graph grammar processing (i.e. all models are reduced to Abstract Syntax Graphs). At the end of the execution of a graph grammar for formalism transformation, the Kernel checks if the resulting model is valid in the active formalism. Formalisms used for intermediate processing are closed appropriately.

5 Meta-Modelling CBD's with AToM^3

As an example of AToM^3's capabilities to model syntax and operational semantics of formalisms, we present Causal Block Diagrams (CBD). CBD are commonly used in tools such as MathWorks' Simulink (tm).

CBDs have two basic entities: *blocks* and *links*. Blocks represent transfer functions, such as arithmetic operators or integators. Links transmit *signals* between blocks. Signals are functions of time. We meta-model CBD syntax by means of an ERD model[1]. Our representation consists of an entity called *block* with an attribute that represents its type[2] (e.g. constant generator, or addition). Links are modelled as a relation between such entities. Links have an attribute representing the value of the signal at the current time of simulation. We also include other elements in our ERD meta-model, called *blinks*, *point*, and *focus*. They will not represent syntactic elements of the CBD per se, but structures necessary to simulate them. This is explained in more detail below. Fig. 3 shows AToM^3 with the CBD meta-model (on the left) and the generated tool to process CBD models (on the right).

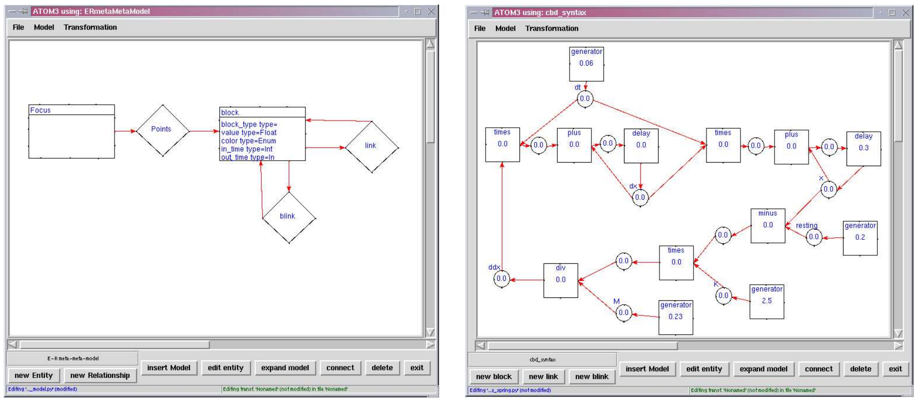

Fig. 3. ER metamodel of CBD (left), and generated tool to process CBD (right)

[1] A meta-meta-model for the ERD formalism is present in AToM^3

[2] AToM^3 has a meta-model of Types.

When simulating CBDs, unless a parallel machine is used, with a processor for each block, where all the processors work in perfect synchronization, one must choose a strategy for propagating information in a way which does not create inconsistencies. This means that there needs to be an ordering in evaluation of dependent nodes. Subgraphs that are independent could be evaluated concurrently, but only before any block that is influenced by them.

The solution is simple: 1) order the nodes by a topological sort of the graph (done by a standard depth-first traversal) 2) evaluate each block following this ordering. In section 6.1 we present such an algorithm by means of a graph grammar. For this we require an additional type of link between blocks, which we call *blink*. A blink between a block B_1 and a block B_2 represents the relation "evaluate B_1 before evaluating B_2". After the topological sort has finished, there will be a hamiltonian path over the blocks where the blocks will be connected by edges of type *blink*. The other entity, the *focus* is a pointer to the block being processed. Only one focus entity is created by the graph grammar, since our approach is purely sequential.

5.1 CBD Denotational Semantics

Here, we provide an informal description of the denotational semantics of block diagrams (Figure 4 on the left). This description simply associates each block diagram to a set of equations representing the values of the links between blocks as signals. More precisely, the denotation of a block diagram is the set of signal functions corresponding to every link in the diagram [3].

In order to simulate CBD on a digital computer we need to discretize the signals, i.e. use the natural numbers as the time-base for the signals. The interpretation of the delay block adopted here is only for a discrete time-base. The other blocks have the same interpretation for both discrete and continuous time. The denotation for the delay block in continuous time has to take a time segment as initial condition, instead of the point value h, shown here.

6 Processing CBD's models

By having the natural numbers as our time base, we can view signals as streams, i.e. unbounded sequences of the values that the signals take at the discrete points. This allows us to see the block diagram as a dataflow network. Here we present an approach based on [1]. One way to model this is by providing each link with an attribute that represents the complete stream computed so far. By doing so, the definition of the operational semantics of CBDs by means of a graph grammar becomes straight-forward. Certaintly this is space-expensive but, as we mentioned in the introduction, the goal of this approach is not to achieve

[3] In our treatment of causal block diagrams we require the explicit use of delay blocks whenever there is a feedback loop. The reason is that otherwise, the denotational semantics given here would produce inconsistencies in the presence of such loops. Furthermore we require there to be at least one constant generator in the model.

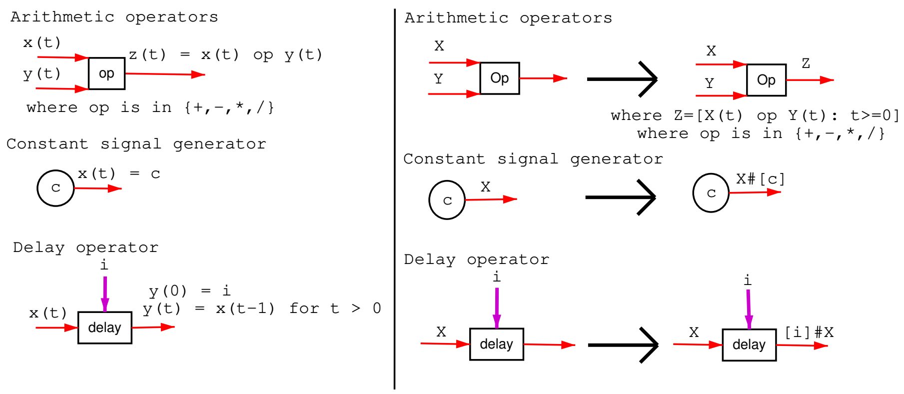

Fig. 4. Denotational (left) and Operational (right) semantics of CBD's.

efficiency, but to be able to define an *executable specification* of the operational semantics.

A first, stream-oriented approach to the operational semantics is straightforward. We observe the following conventions: uppercase letters represent streams, explicit streams are written as lists with square brackets, e.g. $[x_0, x_1, x_2, ...]$. Stream concatenation is done with the $\#$ operator. If we have a finite stream X, then $X\#[e]$ represents the stream resulting from appending e to X. The operational semantics are defined then as shown in figure 4 (right).

This matches the denotational semantics: The constant generator simply generates an infinitely long stream: $X = [c, c, c, ...]$. Hence $X(t) = c$ for all $t \geq 0$. The rule for an arithmetic operator block $*$ guarantee that if $X = [x_0, x_1, x_2, ...]$ and $Y = [y_0, y_1, y_2, ...]$ then $Z = [x_0 * y_0, x_1 * y_1, x_2 * y_2, ...]$, that is, $Z(t) = X(t) * Y(t)$ for all $t \geq 0$. Finally, for the delay operator we have that if the input is $X = [x_0, x_1, x_2, ...]$ then the output is $Y = [i, x_0, x_1, x_2, ...]$, i.e. $Y(0) = i$, $Y(1) = X(0)$, $Y(2) = X(1)$, etc. Hence $Y(t) = X(t-1)$ for all $t > 0$.

6.1 Topological Sort

The problem with these rules is that they do not take into account the issue of evaluation order. This might be enough to reason about CBDs, but not to produce an "executable specification". In order to deal with this, we introduce a set of rules which will sort the blocks. This set of rules is to be evaluated before the actual operational semantics rules.

The general idea of this set of rules is based on a depth-first search of the graph [3]. Here we explicitly construct the evaluation path, i.e. we create blinks between the blocks. First we find some root block[4], a block without parents, and

[4] In the CBD presented here there will always be at least one root node, since we require to be at least one constant generator.

visit all its children recursively, marking with a colour blocks already visited. When a node has not been visited, it is white. When it has been visited but not all its decendants have been explored, it is gray. Otherwise it is black. We also keep a pointer to the node currently visited, which we call the *focus*. As the focus goes from parent to child, a *blink* edge is created between them, and a blink coming out of the parent is transferred to the child. When backtracking after finding a dead-end (i.e. a gray or black node, already visited), blinks are left unchanged. When a branch has been completely explored, a new root is searched for and the process is repeated until all nodes are coloured black.

Given the space limitations only some representative rules are shown (for details, we refer to the AToM^3 web-page [2]). Blocks will have a counter representing the number of immediate children being explored. The rule in Fig. 5 shows the rule representing the discovery of a node that hasn't been visited, as described above. (Dashed arrows are blinks.)

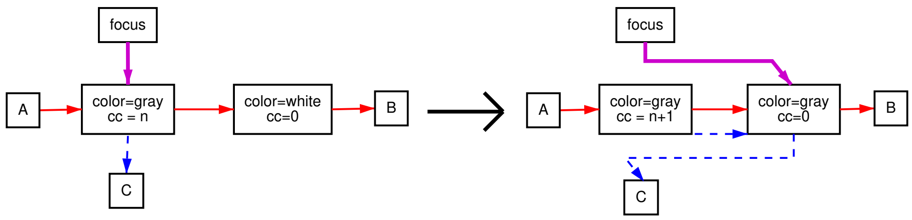

Fig. 5. Topological sort of a CBD: new non-terminal discovered

Another important rule is shown in Fig. 6, depicting the backtracking when a loop is detected or when all the children have been visited.

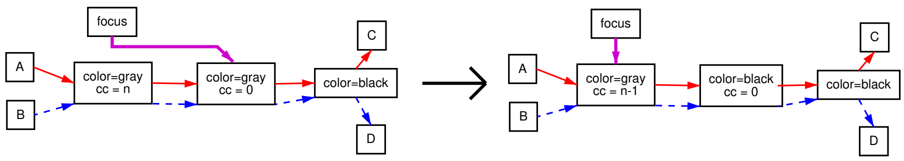

Fig. 6. Topological sort of a CBD: loop detected or children completed.

The graph-grammar implementing the topological sort, adds a blink between the last node and the first, making it a loop.

6.2 Operational Semantics

We need to adapt the rules shown in Fig. 4 so that blocks are evaluated in the correct order. This is implemented by focusing on one block, evaluating it, and follow the blinks created by the topological sort. An example of one such rule is shown in figure 7.

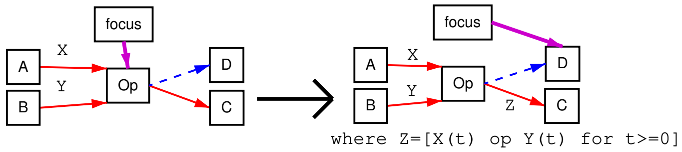

Fig. 7. A representative rule for evaluation of a CBD.

Since the topological sort returns a loop covering all nodes, evaluation proceeds following the described scheme until some termination criteria is met. To specify termination, we add two global attributes to the meta-model of CBD: an iteration counter, and a maximum number of iterations attribute. Models in AToM^3 can also have user-defined constraints, making it easy to define the termination criteria in terms of these atributes.

6.3 Simulation Results

The specified CBD simulator was tested on the harmonic oscillator equation (also known as the "circle test"): $\frac{dx^2}{dt^2} = -x, x(0) = 1, \frac{dx}{dt}(0) = 0$. Full results can be found on the $AToM^3$ homepage [2].

7 Conclusions

In this article we have presented our approach to modelling complex systems, which is based on meta-modelling and multi-formalism modelling, and is implemented in the software tool AToM^3. This code-generating tool, developed in Python, relies on graph grammars and meta-modelling techniques.

We have demonstrated how both syntax and operational semantics of the commonly used formalism Causal Block Diagrams formalism can be modelled. When doing so in AToM^3, a tool for modelling and simulating CBD is automatically obtained.

Our main contribution is the unification of meta-modelling (formalisms – classes of models – may be modelled in their own right) and graph transformation based on graph grammar specifications.

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 kinds of models we will deal with. The processing of such models can be expressed at the meta-level by means of graph grammars.

AToM^3, with meta-models for modelling with Entity-Relationship, Data Flow Diagrams, Structure Charts, Petri-Nets, Statecharts, GPSS, DEVS and Finite State Automata and some transformations is available at [2].

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.

References

1. Abelson H., Sussman G. J. *Structure and Interpretation of Computer Programs* 2nd edition. MIT Press. 1996.
2. AToM^3 Home page: http://moncs.cs.mcgill.ca/MSDL/research/projects/ATOM3.html
3. Cormen, T., Leiersson, C. H., Rivest, R. S. *Introduction to Algorithms* 1st edition. MIT Press. 1990.
4. DOME guide. http://www.htc.honeywell.com/dome/, Honeywell Technology Center. Honeywell, 1999, version 5.2.1
5. Dorr, H. 1995. *Efficient Graph Rewriting and its implementation*. Lecture Notes in Computer Science, 922. Springer.
6. 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. Murata, T. *Petri Nets: Properties, analysis and applications.* Proceedings of the IEEE, 77(4)-541-580.
8. OMG Home Page: *http://www.omg.org*
9. Python home page: *http://www.python.org*
10. Sztipanovits, J., et al. 1995. *"MULTIGRAPH: An architecture for model-integrated computing"*. In ICECCS'95, pp. 361-368, Ft. Lauderdale, Florida, Nov. 1995.
11. Vangheluwe, H. *DEVS as a common denominator for multi-formalism hybrid systems modelling.* In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134. IEEE Computer Society Press, September 2000. Anchorage, Alaska.
12. Zeigler, B., et al. 2000 *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press. Second Edition. 2000