Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

META-MODELS ARE MODELS TOO

Hans Vangheluwe

School of Computer Science
McGill University
Montréal, Québec H3A 2A7, CANADA

Juan de Lara

ETS Informática
Universidad Autonóma de Madrid
Madrid, SPAIN

ABSTRACT

This article introduces *multi-formalism* modelling and *meta-modelling* to facilitate computer assisted modelling and simulation of complex systems. To aid in the automatic generation of multi-formalism modelling and simulation tools, formalisms are modelled in their own right, at a meta-level, within an appropriate formalism. This approach is implemented in the interactive tool ATOM^3 (A Tool for Multi-formalism Meta-Modelling). This tool is used to describe formalisms commonly used in the simulation of dynamical systems, as well as to generate custom tools to process (create, edit, simulate, …) models expressed in the corresponding formalism. ATOM^3 relies on graph rewriting techniques to perform the transformations (modelled as graph grammars) between formalisms as well as for other tasks, such as code generation or simulator specification.

The Finite State Automata (FSA) formalism is used to demonstrate the concepts of meta-modelling as well as model transformation (in particular, simulation of FSA models).

The issue of a neutral model exchange and re-use format is addressed in the context of meta-modelling. Core XML is proposed as a standard external format. Thanks to the power of the meta-modelling approach, DTD, XMLSchema, and XSLT specifications may be replaced by models, externally represented in core XML, in appropriate formalisms (Entity Relationship for syntax and Graph Grammar for transformation respectively).

1 INTRODUCTION

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

- A single super-formalism may be constructed which subsumes all the formalisms needed in the system description. This is not possible nor meaningful in most cases, although there are some examples of formalisms which span several domains (*e.g.*, Bond Graphs for the mechanical, hydraulic and electrical domains.)
- Each system component may be modelled using the most appropriate formalism and tool. To investigate the overall behaviour of the system, co-simulation can be used. In this approach, each component model is simulated with a formalism-specific simulator. Interaction due to component coupling is resolved at the trajectory (simulation data) level. The co-simulation engine orchestrates the flow of input/output data in a data-flow fashion. In this approach, questions about the overall system can only be answered at the level of input/output (state trajectory) level. It is no longer possible to answer higher-level questions which could be answered within the formalisms of the individual components. Furthermore, there are speed and numerical accuracy problems for continuous formalisms, in particular if one attempts to support computationally non-causal models. The co-simulation approach is meaningful mostly for discrete-event formalisms. It is the basis of the DoD High Level Architecture (HLA) `<hla.dmso.mil>` for simulator interoperability.
- In multi-formalism modelling, as in co-simulation, each system component may be modelled using the most appropriate formalism and tool. However, a single formalism is identified into which each of the component models may be symbolically transformed (Vangheluwe 2000). Obviously, the system properties which we wish to investigate

must be invariant under the transformations. The formalism to transform depends on the question to be answered about the system. The Formalism Transformation Graph (see Figure 1) suggests DEVS (Zeigler, Praehofer, and Kim 2000) as a universal common modelling formalism for simulation purposes (generating input/output trajectories).

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

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

In this article, we present ATOM^3, a tool which implements the ideas presented above. Using Finite State Automata as an example, we demonstrate the use of meta-modelling to specify and automatically generate an FSA modelling and simulation tool. To allow tool-neutral exchange and re-use of models, XML is proposed as an external model storage format. As meta-models are models too, there is no need for a dedicated syntax such as DTD or XMLSchema to describe formalism syntax.

2 MULTI-FORMALISM MODELLING

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

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

3 META-MODELLING

As stated in the previous section, one of the characteristics of complex systems is the diversity of their components. Consequently, it is often desirable to model the different components using different modelling formalisms. This is certainly the case, when inter-disciplinary teams collaborate on the development of a single system. A proven method to achieve the required flexibility for a modelling language that supports many formalisms and modelling paradigms is to model the modelling language itself (Honeywell 1999). Such a model of the modelling language is called a meta-model. It describes the possible structures which can be expressed in the language. A meta-model can easily be tailored to specific needs of particular domains. This requires the meta-model modelling formalism to be rich enough to support the constructs needed to define a modelling language. Taking the methodology one step further, the meta-modelling formalism itself may be modelled by means of a meta-meta-model. This meta-meta-model specification captures the basic elements needed to design a formalism. Table 1 depicts the levels considered in our meta-modelling approach. Formalisms such as Entity-Relationship diagrams 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 State Automaton, different transitions leaving a given state must have different labels. This cannot be expressed within Entity-Relationship diagrams alone. Expressing constraints is most elegantly done by adding a constraint language to the meta-modelling formalism. Whereas the meta-modelling formalism frequently uses a graphical notation, constraints are concisely expressed in textual form. For this purpose, some systems (Gray, Bapty, and Neema 2000), including ATOM^3 use the Object Constraint Language OCL <www.omg.org> used in the UML.

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

3.1 Meta-Modelling FSA in ATOM^3

ATOM^3 is a tool written in Python <www.python.org> which uses and implements the meta-modelling concepts presented above.

The main component of ATOM^3 is the Kernel. This module is responsible for loading, saving, creating and manipulating models, as well as for generating code. By default, a meta-meta-model is loaded when ATOM^3 is invoked. This meta-meta-model allows us to model meta-

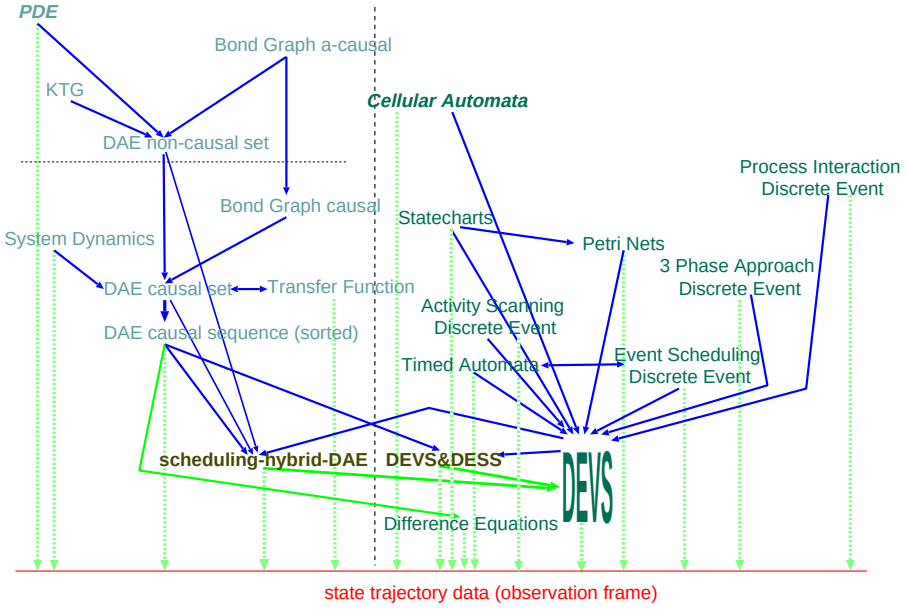

Figure 1: Formalism Transformation Graph

Table 1: Meta-Modelling Levels

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

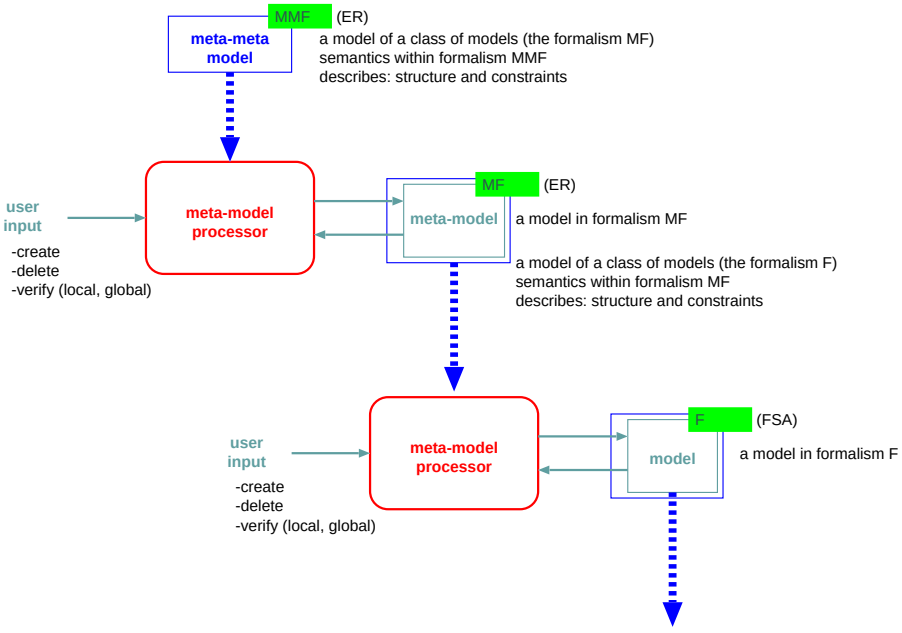

Figure 2: Proposed Working Scheme for a Meta-Modelling Environment

models (modelling formalisms) using a graphical notation. For the moment, the Entity-Relationship formalism extended with constraints is available at the meta-meta-level. When modelling at the meta-meta-level, the entities which may appear in a model must be specified together with their attributes. We will refer to this as the semantic information. For example, to define the Deterministic Finite Automaton Formalism, it is necessary to define both States and Transitions. Furthermore, for States we need to add the attribute name and type (initial, terminal or regular). For Transitions, we need to specify the condition that triggers it. This is shown in Figure 3. Note how the "current" entity is present to facilitate model simulation.

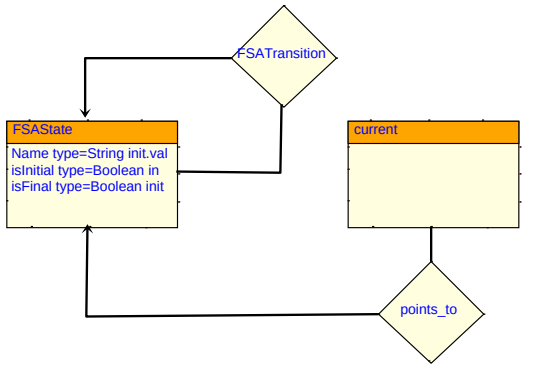

Figure 3: FSA Meta-Model (ER+Constraints)

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

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

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

Using the FSA meta-model information, ATOM^3 allows editing of syntactically correct FSA models. As an example, an FSA recognizing even binary numbers (the input is a sequence of 0s and 1s) is shown in Figure 4. Actually, this figure already shows the first step of a graph grammar specified FSA simulator (asking the user for the input sequence).

3.2 Graph Grammars

Both for modelling and simulation purposes it is necessary to transform (structure as well as attributes) of models. The Graph Grammars formalism (Dorr 1995) allows us to *model* model transformations. In analogy to string grammars, graph grammars can be used to describe graph transformations, or to generate sets of valid graphs. Graph grammars are composed of rules, each mapping a graph on the left-hand side to a graph on the right-hand side. When a match is found between the left-hand side of a rule and a part of an input graph, this subgraph is replaced by the right-hand side of the rule. Rules may also have a condition that must be satisfied in order for the rule to be applied, as well as actions to be performed when the rule is executed. A rewriting system iteratively applies matching rules in the grammar to the graph, until no more rules are applicable.

The use of a model (in the form of a graph grammar) of graph transformations has some noteworthy advantages over an implicit representation (embedding the transformation computation in a program) (Blonstein, Fahmy, and

Grbavec 1996). The main advantages using a model of graph transformations can be summarized as follows:

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

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

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

In a graph grammar, entities are labelled with numbers. In our case, entities are both states and transitions. If two nodes in a left hand side and a right hand side have the same number, that means that the node must not disappear when the rule is executed. If a number appears in a LHS but not in a RHS, that means that the node must be removed when applying the rule. If a number appears in a RHS but not in a LHS, that means that the node must be created if the rule is applied.

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

Here, we describe the operational semantics of the FSA formalism by means of a Graph Grammar model. Executing the Graph Grammar on a particular FSA model and input segment will result in a simulation trace. The graph grammar is depicted in Figure 5. The user constructs an FSA model in AToM^3 as shown in the even binary number recognizer model of Figure 4. As an initial action, the user is prompted to provide an input segment ("010" in this case). Rule 1 identifies the initial state of the model, creates a "current" node and lets it point to the initial state. Subsequently, Rules 2 and 3 "match" a transition pattern (between different states or in a self-loop respectively) and the current input and re-write the model appropriately. The simulation continues until no more rules match. This process is depicted in Figure 6 for the even binary number recognizer FSA example with "010" as input segment.

4 XML REPRESENTATION

By default, AToM^3 saves models as Python scripts. This removes the need to parse the model as this is taken care of by the Python interpreter. Formalism-specific syntax checks are done upon reading. They are based on information from the (ER + constraints) meta-model of the formalism the model being loaded is expressed in.

For model exchange and re-use purposes (possibly over the WWW), a neutral format is required. XML seems the most viable candidate. XML is a base-language for expressing arbitrary, structured data in text form. It consists of several modules: core syntax, meta-syntax, linking, style-bindings, …Of these only the core syntax is common to all XML applications. Applications can choose to omit the other modules if they don't need them. All attributes in XML are by default string valued, although the meta-syntax is able to restrict that.

As depicted in Figure 7, it is sufficient to hand-craft a lexical analyser and parser for "core XML" syntax as described by Bert Bos in `<http://www.w3.org/XML/9707/XML-in-C>`.

The core XML grammar in pseudo-EBNF is given below:

```
document: prolog element misc*;
prolog: VERSION? ENCODING? misc*;
misc: COMMENT | attribute_decl;
attribute_decl: ATTDEF NAME attribute+ ENDDEF;
element: START attribute* empty_or_content;
empty_or_content:
 SLASH CLOSE | CLOSE content END NAME? CLOSE;
content: (DATA | misc | element)*;
attribute: NAME (EQ VALUE)?;
```

Literals are appropriately defined in the lexical specification `<http://www.w3.org/XML/9707/scanner.l>`. The full Bison specification is available at `<http://www.w3.org/XML/9707/parser.y>`

The core XML syntax accepts any nested combination of `<tag> ...</tag>` or `< ... />` constructs. Rather than using DTDs or XMLSchema to specify formalism-specific valid syntax, we propose a more expressive ER+constraints model. Note how such a meta-model (of formalism F) is a model in its own right and can be saved in XML format. One a model's syntax has been checked, it can be transformed (simplified, simulated, converted into another formalism, …). Transformations are again modelled, most likely in the Graph Grammar formalism. Transformation models can again be saved and shared in XML format. Note how transformation specifications such as XSLT are

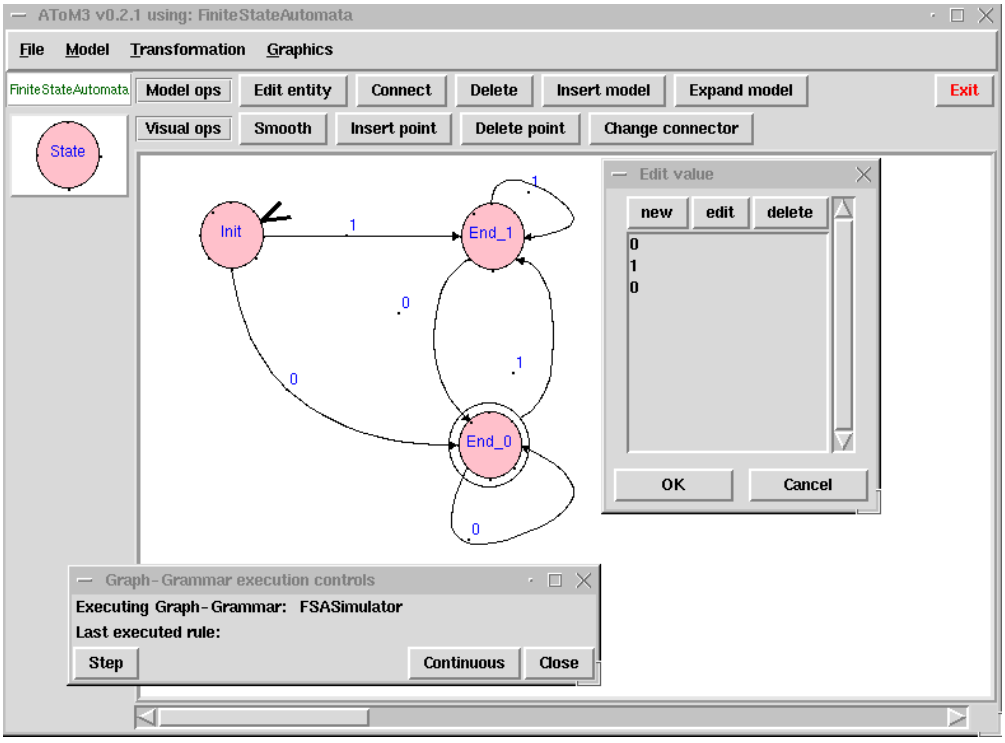

Figure 4: FSA Model, Binary Digit Sequence Input

easily expressed in this framework. Ultimately, we need to save the model under study in the neutral XML format. Output generation is nothing but another transformation and can as such again be explicitly modelled. The whole scenario is depicted in Figure 7. The approach can easily be extended to read and write custom modelling languages such as Modelica (Elmqvist et al. 1999).

5 CONCLUSIONS

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

The advantages of using such an automated tool for generating customized model-processing tools are clear: instead of building the whole application from scratch, it is only necessary to specify – in a graphical manner – the kinds of models we will deal with. The processing of such models can be expressed 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 (one that already has a simulator available, for example). We then obtain a tool to model in the new formalism, and are able to convert models in this formalism into the other for further processing.

A side effect of our code-generating approach is that some parts of the tool have been built using code generated by itself (bootstrapped): one of the first implemented features of AToM^3 was the capability to generate code, and extra features were added using code thus generated.

As there is a need for a neutral model exchange format, we have presented XML and its integration in the meta-modelling framework.

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

ACKNOWLEDGMENT

Prof. Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant.

REFERENCES

Blonstein, D., H. Fahmy, and A. Grbavec. 1996. Issues in the practical use of graph rewriting. *Lecture Notes in Computer Science* 1073:38–55.

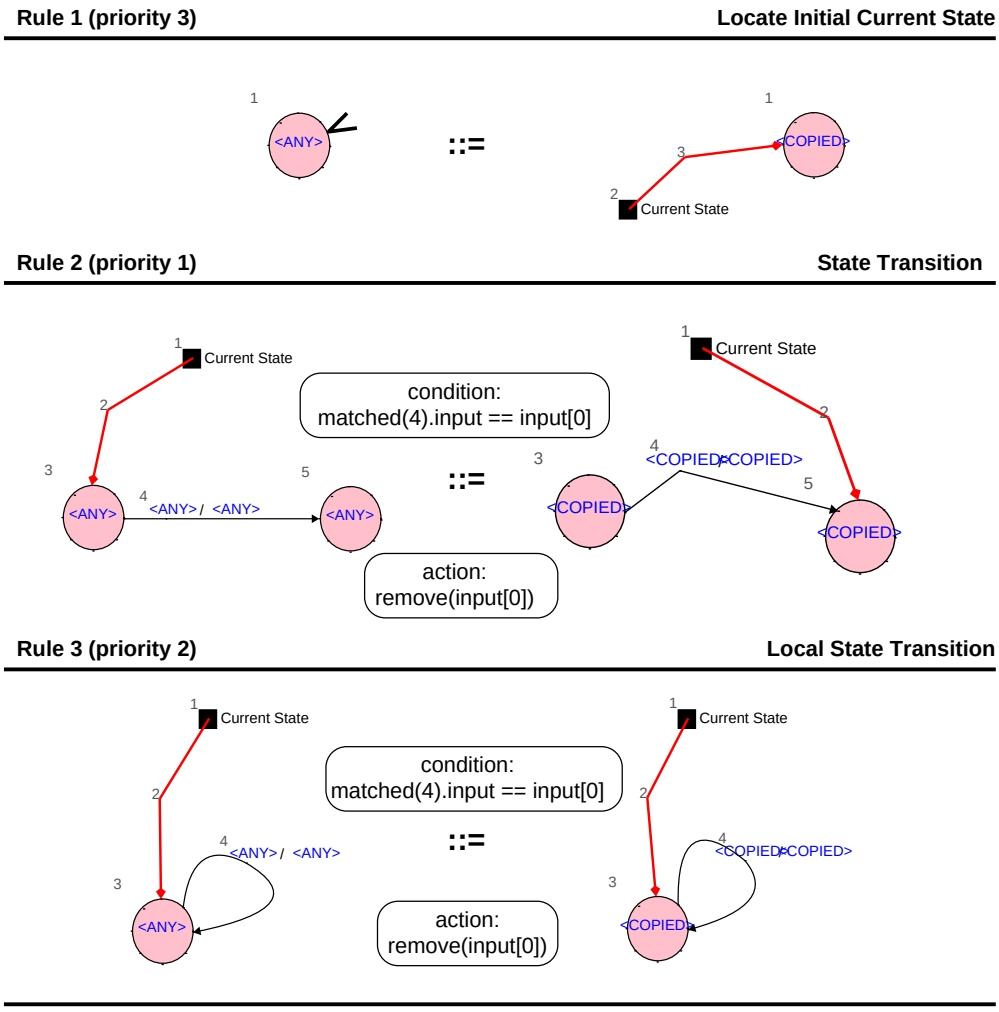

Figure 5: FSA Simulator Model (GG)

Dorr, H. 1995. Efficient graph rewriting and its implementation. *Lecture Notes in Computer Science* 922.

Elmqvist, H., B. Bachmann, F. Boudard, J. Broenink, D. Brück, T. Ernst, R. Franke, P. Fritzson, A. Jeandel, P. Grozman, K. Juslin, D. Kågedal, M. Klose, N. Loubère, S.-E. Mattson, P. Mosterman, H. Nilsson, M. Otter, P. Sahlin, A. Schneider, H. Tummescheit, and H. Vangheluwe. 1999. Modelica – a unified object-oriented language for physical systems modeling: Tutorial and rationale. Report, The Modelica Design Group. `<http://www.modelica.org/>`.

Gray, J., T. Bapty, and S. Neema. 2000. Aspectifying constraints in model-integrated computing. In *OOPSLA Workshop on Advanced Separation of Concerns*. Minneapolis, MN.

Honeywell 1999. DOME guide. `<http://www.htc.honeywell.com/dome/>`. Honeywell Technology Center, Honeywell. version 5.2.1.

Vangheluwe, H. L. 2000. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design*, ed. A. Varga, 129–134: IEEE Computer Society Press. Anchorage, Alaska.

Vangheluwe, H. L., J. de Lara, and P. J. Mosterman. 2002. An introduction to multi-paradigm modelling and simulation. In *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, ed. F. Barros and N. Giambiasi, 9 – 20. Lisboa, Portugal.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modelling and simulation: Integrating discrete event and continuous complex dynamic systems*. Second ed. Academic Press.

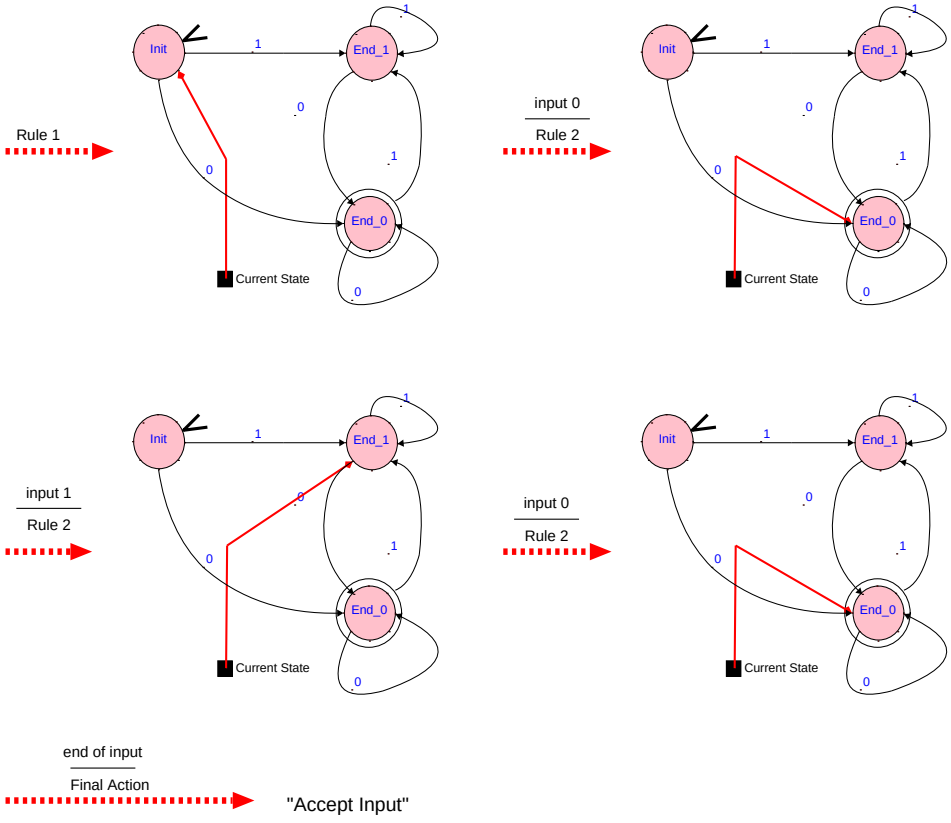

Figure 6: Simulation Steps (Rule Matches Due to Input)

AUTHOR BIOGRAPHIES

HANS VANGHELUWE, D.Sc. 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 (MSDL). He has been the Principal Investigator of a number of research projects on the development of a multi-formalism theory for Modelling and Simulation. Some of this 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 current research is focused on the development of the AToM^3 tool for Computer Aided Multi-Paradigm Modelling (CAMPaM) and simulation. His e-mail address is `<hv@cs.mcgill.ca>`, and his web page is `<www.cs.mcgill.ca/~hv>`.

JUAN DE LARA is an Assistant Professor at the Universidad Autonóma (UAM) de Madrid in Spain, where he teaches software engineering, as well as modelling and simulation. His research interests include Web Based Simulation, Meta-Modelling, distance learning, and social agents. He recieved his PhD in June 2000 at UAM. He graduated in 1994 with a Top of Class Award as a Technical Engineer in Computer Science. In 1996 he received the honour of Higher Engineer in Computer Science. During 2001, as a post-doctoral researcher in the MSDL, he created the AToM^3 prototype. His e-mail address is `<Juan.Lara@ii.uam.es>`, and his web page is `<www.ii.uam.es/~jlara>`.

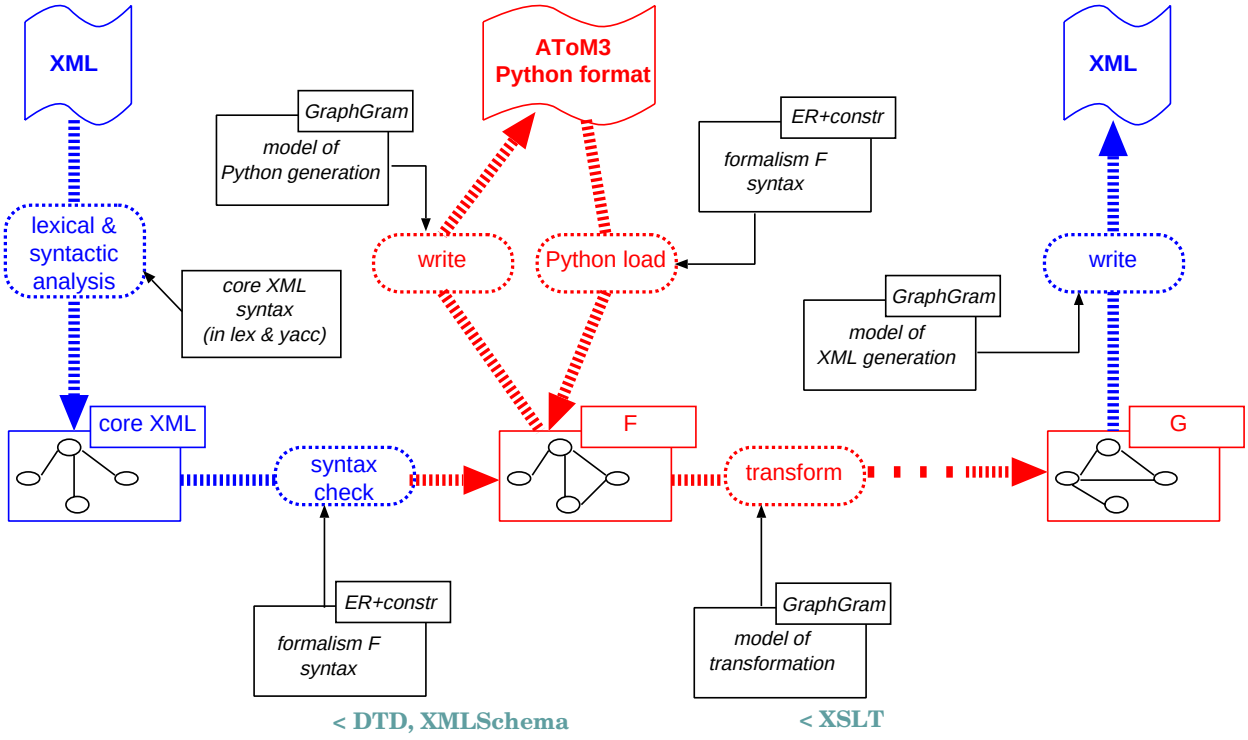

Figure 7: XML and Meta-Modelling