Using Meta-Modelling and Graph Grammars to process GPSS models

Juan de Lara[1,2]
[1]E.T.S. de Informática
Universidad Autónoma de Madrid
Ctra. Colmenar km. 15, Campus Cantoblanco
28049 Madrid, Spain
Juan.Lara@ii.uam.es, jlara@cs.mcgill.ca

Hans Vangheluwe[2]
[2]School of Computer Science
McGill University
3480 University Street
Montréal, Québec, Canada H3A 2A7
hv@cs.mcgill.ca

Abstract

This paper discusses the benefits of combining meta-modelling and graph transformations to automatically generate modelling tools for simulation formalisms. In meta-modelling, formalisms are modelled in their own right at a meta-level within an appropriate meta-formalism. A meta-model processor uses this information to automatically *generate tools* to process –create, edit, check, optimize, transform and generate simulators for– the models in the described formalism. We propose the representation of (meta-)models as graphs, and subsequently specify model manipulations as graph grammars. We also present AToM^3, A Tool for Multi-formalism and Meta-Modelling which implements these concepts. As an example, we show how to build a meta-model for the popular process interaction discrete event language GPSS in AToM^3. From this meta-model, AToM^3 automatically generates a visual tool to build GPSS models. We also define a graph grammar to generate textual code for the HGPSS simulator from the graphically specified GPSS models.

Keywords: Meta-Modelling, Multi-Formalism, Graph Grammars, GPSS, Automatic Code Synthesis.

1 Introduction

Meta-Modelling is the process of modelling formalisms. In the context of Modelling and Simulation we are interested in formalisms such as Petri nets, DEVS, GPSS [11] [23] and Ordinary Differential Equations (ODEs).

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

In Modelling and Simulation we are also interested in other model manipulations, such as:

1. Model simulation.
2. Model optimization, for example, reducing its complexity.
3. Model transformation into another (behaviourally equivalent) model, expressed in a different formalism. This may be useful in several situations:
   - In a composite model with multiple heterogeneous components, it may be possible to transform each of those components into a common formalism. Then, once the composite model has all its components described in the same formalism, it is possible to process the entire model meaningfully. This is the core of the multi-formalism approach to modelling complex systems [25].
   - To solve problems that are easier in another formalism. An example is the case of an ODE model which does not have an analytical solution. To solve this problem we transform the model into the Difference Equations formalism (this is usually done by numerical solvers) and solve it in this domain. We will lose some information when performing this particular transformation, due to discretization.
4. Generation of code for an existing simulator.

In this paper, we focus on the application of model transformation.

We present AToM^3, a tool which implements the ideas presented above. AToM^3 has a meta-modelling layer in which different 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*. As a consequence, model manipulation such as the ones listed above can be expressed as graph grammars [9].

As an example, we show the generation of a tool to graphically manipulate GPSS models. We also define a graph grammar to generate textual representations of the graphical models that can be run on GPSS simulators. The simulation community has many efficient GPSS simulators available. Many of them have been criticised for their lack of a graphical interface for modelling, though from its inception, GPSS has had a standard graphical representation for its blocks. Arguably, this is one of the main reasons for the success of GPSS.

Different GPSS simulators implement slight variations of the original GPSS. A meta-modelling approach is thus very suitable, as one can generate tools for small variants of the formalism with little effort. A meta-modelling approach also makes it possible to devise graphical representations for elements of the model for which there is no standard graphical representation, such as storages and table sizes, functions and variables. Of course, thanks to the flexibility of meta-modelling, these representations can be readily modified. In any case, a graph grammar for textual code generation can dump and rearrange this graphical information into a text file for further processing by a GPSS simulator such as our HGPSS [6].

2 Meta-Modelling

Meta-Modelling is the process of explicitly modelling syntax and semantics of formalisms. Any model has meaning within the context of a formalism. As meta-models describe classes of models, the formalism in which meta-models are described (known as the meta-formalism) neeeds to be expressive enough. Typically, the Entity Relationship diagrams or UML class diagram [20] formalisms are used. A model of a formalism is called a meta-model. A model of a meta-formalism is called a meta-meta-model. Table 1 depicts the levels considered in our meta-modelling approach. Note that we only consider three levels. When a meta-formalism mf_1 is powerful enough to describe the meta-meta-model of another meta-formalism mf_2, we consider both mf_1 and mf_2 as meta-formalisms and place them at the same meta-level. The meta-formalisms currently used in AToM^3 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 Determinisitic Finite Automata, different transitions leaving a given state must have distinct labels. This cannot be expressed within Entity Relationship diagrams alone. Expressing constraints is most elegantly done by extending the meta-formalism with a constraint language. Whereas the basic meta-formalism frequently uses a graphical notation, constraints are concisely expressed in textual form. For this purpose, some systems [14] (including ours) use the Object Constraint Language OCL [20] used in UML. As AToM^3 [4] is implemented in the scripting language Python [22], arbitrary Python code can also be used.

Level	Description	Examples
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 in a formalism. 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.

3 Graph Grammars: an introduction

To completely describe a formalism, not only the syntax (structure, described in the meta-model) needs to be specified, but also the semantics. One way to describe semantics is by explicitly specifying transformations on models in the formalism.

In our approach, we store models as (hyper)graphs called Abstract Syntax Graphs (ASGs), and thus we use graph grammars to express model manipulation. Graph grammars are composed of tranformation rules. Each rule maps a graph on the left-hand side (LHS) onto a graph on the right-hand side (RHS). A graph grammar is applied to an input graph (called host graph) to perform a transformation. When a match is found between the LHS of a rule and a part of the host graph, this matching subgraph is replaced by the RHS of the rule. Rules may also have conditions which 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 [9]. Some approaches also offer control flow specifications. In AToM^3, rules are given a priority and are tried in ascending order.

On the one hand, the use of a model (in the form of a graph grammar) of graph transformations has some advantages over an implicit representation (embedding the transformation computation in a program) [5]:

- It is an abstract, declarative, high level representation. As it is a model in its own right, it may be used as a basis for analysis, transformation, etc.
- The theoretical foundations of graph rewriting systems can 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 isomor-

phism testing is NP-complete. However, using small subgraphs on the left hand side of graph grammar rules, as well as node labels and edge labels can greatly reduce the search space.

4 AToM^3: an overview

AToM^3 [7] is a tool written in Python [22] which implements the concepts presented above. Its architecture is shown in Figures 1 and 2. In both figures, models are represented as white boxes, having on their upper-right hand-corner an indication of the meta-... model (formalism) they are specified in. In the case of a graph grammar model, to convert a model in formalism F_{source} to F_{dest}, it is necessary to use the meta-models of both F_{source} and F_{dest} in addition to the meta-model of graph grammars.

The main component of AToM^3 is the AToM^3 Kernel, which is responsible for loading, saving, creating and manipulating models (at any meta-level, by means of the Graph Rewriting Processor), as well as for generating code for custom tools. Both meta-models and meta-meta-models can be loaded into AToM^3 as shown in Figure 1. The first kind of models is used to construct valid models in a certain formalism, the second is used to describe the formalisms themselves.

In AToM^3, the Entity-Relationship (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 the ER formalism. 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. We will refer to this as the semantic information. For example, to define the Petri Net Formalism, it is necessary to define both *Places* and *Transitions*. Furthermore, for *Places* we need to add the attributes *name* and *number of tokens*. For *Transitions*, we need to specify the *name*.

The meta-meta-model is used by the AToM^3 Kernel to generate some Python files, which, when loaded by the Kernel, allow for processing of models in the defined formalism. One of the components of the generated files is a *model* of a part of the AToM^3 user interface. This User Interface model has meaning in the Buttons formalism which has its own meta-model. Initially, this model represents the necessary GUI buttons to interactively create the entities defined in the formalism's meta-model, but can be modified to include for example, buttons to execute graph grammars on the current model. We give an example of modifying the AToM^3 User Interface Model in the next section.

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.

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. For example, for Petri Nets, we can choose to represent *Places* as circles with the *number of tokens* inside the circle and the *name* beside them, and *Transitions* as thin rectangles with the *name* beside them. That is, we can specify how some semantic attributes are displayed graphically. Constraints can also be associated with the graphical entities. Each part of the graphical entity can be referenced by an automatically generated name which gives access to methods to change the part's colour, or to hide it.

AToM^3 allows for explicit modelling of types. In particular, it is possible to specify *composite* types. These are defined by constructing a type graph [3]. The Meta-model for this graph was built using AToM^3 and then incorporated into the AToM^3 Processor. The components of this graph can be basic or composite types and can be combined using the *product* and *union* type operators. Types may be recursively defined, meaning that one of the operands of a *product* or *union* operator can be an ancestor node. Infinite recursive loops are detected using a global constraint in the type meta-model. The graph describing the type is compiled into Python code using a graph grammar (also defined using AToM^3).

The AToM^3 source with a collection of useful meta-models can be found at [4].

5 Meta-Modelling GPSS

The process interaction discrete event simulation language GPSS was described by Gordon in the early 1960s [11] [23]. The simplicity and universality of GPSS concepts as well as the easy-to-learn block diagram notation have made it a popular choice for modelling and simulation. An excellent overview of 40 years of GPSS is given in [12].

In this section, we show how to generate a visual tool using AToM^3 for GPSS modelling.

In our GPSS meta-model the basic entity is the *GPSSBlock*. In the GPSS formalism there are a number of different blocks (the number depends on the GPSS flavour), but usually there are about twenty. Each of these blocks have a different semantics, and one can construct a GPSS model by connecting these blocks. Additional information must be provided textually. Thus, *GPSSBlock*s have a *type* and an associated graphical icon that changes depending on this *type*. This change is performed by a post-action on the *EDIT* event. *GPSSBlocks* also have a number of parameters (named A, B, ...,F) whose semantics depends on the type of the block, a *Label* and a field to include comments. The

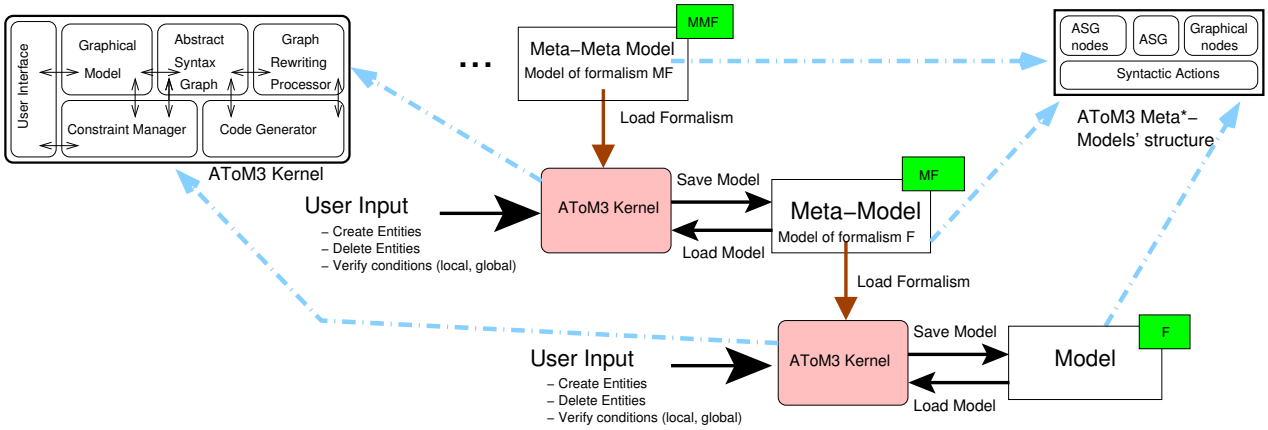

Figure 1: Meta-... Modelling in AToM^3.

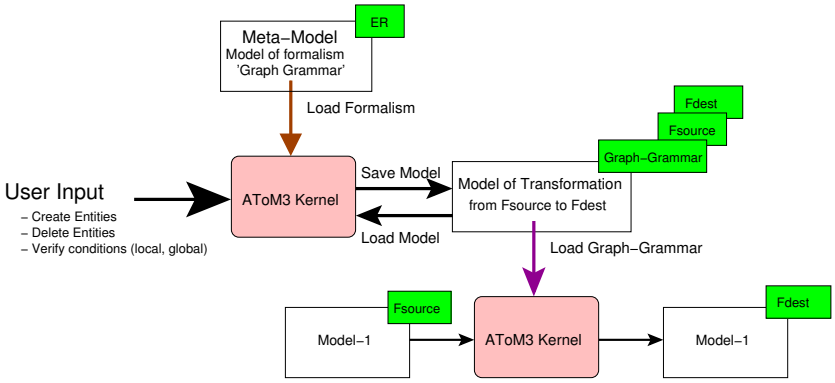

Figure 2: Model Transformation in AToM^3.

Label is filled automatically with a unique string when the entity is created, but can be modified by the user.

*GPSSBlock*s can be connected to other *GPSSBlock*s. Constraints have been added to regulate the number of outgoing connections a block can have (some may have one, while others may have two). Some of the blocks which have two outgoing connections store the label of the connected entities in parameters *A* and *B*. These parameters are filled appropriately by means of post-actions on the *CONNECT* event.

Other entities have been defined in the meta-model to describe table properties, functions, variables and storage sizes. Each of these entities, which cannot be connected, have been given a visual representation (and icon). The model as a whole has been provided with some global attributes, namely the model and author names, and a list of control statements (the simulation experiment description). The latter are implemented as a composite type, which is composed of the statement name (an enumerate type) and some additional parameters.

On the left hand side of Figure 3, AToM^3 uses the Entity Relationship formalism to describe the GPSS meta-model. The generated GPSS modelling tool is shown on the right hand side.

As explained in the previous section, AToM^3 generates a model of the user interface (in the Buttons formalism). The Buttons meta-model (in the ER formalism) is composed of one entity: the *Button* entity. This entity has attributes to edit the text or image of the button, and to add an action in the form of Python code. By default, a User Interface Model is generated with one button for each *entity* or *relationship* in the meta-model. To obtain the tool of the right hand side of figure 3, we have modified this model with AToM^3, as the default User interface model contained buttons for the defined entities (the *GPSSBlock*, *Table*, *Function*, *SNA Variables* and *Storage Sizes*) and the *connected_to* relationship. We have deleted the button for this last relationship and added a button *Gen.Code* to execute the graph grammar to generate textual code for the HGPSS simulator. This graph grammar is explained in the next section.

6 Processing GPSS models

In this section, we construct a graph grammar which takes GPSS models defined using the tool generated in the pre-

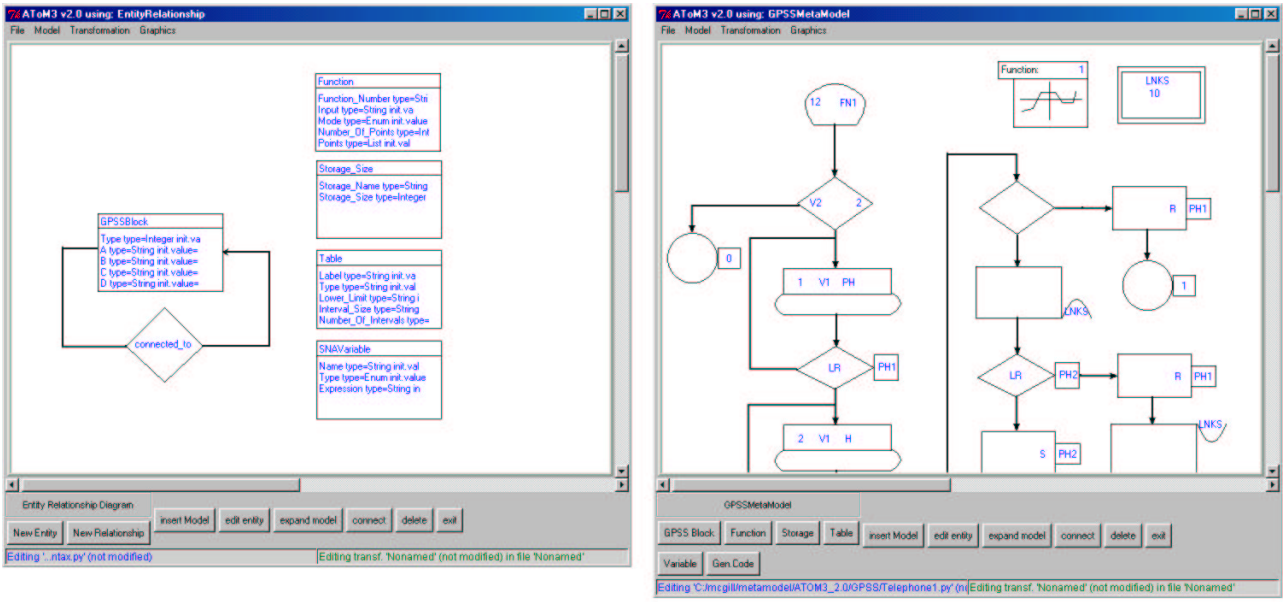

Figure 3: GPSS Meta-Model and Generated Tool to Process GPSS Models.

vious section, and produces textual code to be executed in a GPSS simulator. This graph grammar is shown in Figure 4. We can identify two types of rules: those ones dealing with the *GPSSBlocks* (rules 2-6), and those dealing with functions, tables, storages, and variable definitions (rules 1, 7-9).

The graph grammar has an *initial action* which opens the file where the code is to be generated and decorates all the entities in the model with two auxiliary attributes: *current* and *visited*. The *current* attribute is used to identify the node in the model whose code has to be generated next. The *visited* attribute is used to determine whether code for the node has been generated yet.

The graph grammar is composed of nine rules which are tried in ascending order:

Rule 1 locates function descriptions and generates the corresponding code.

Rule 2 locates *GATE* or *TEST* blocks and generates code for them. These blocks have two children. The rule has to ensure that the branch corresponding to the *A* parameter is visited first.

Rule 3 traverses the selected branch in a *depth first* way, generating code for each visited node.

Rule 4 is executed when a branch ends. Note that the *current* attribute is not transferred to a new node. Rather, rule 5 will locate a new branch to traverse.

Rule 5 starts a new branch, setting the *current* attribute of the next node to be visited.

Rule 6 is executed when a root node is found. Root nodes do not have any incoming connections, and the code generation of the *GPSSBlocks* graph starts in them. Caveat: GPSS allows incoming connections to a GENERATE block. This may lead to models without root node, a case not covered in our rules. To cover this pathological case, it suffices to use a generate block as the root node.

Remaining rules generate code for tables, storages and variables definitions.

The graph grammar also has a *final action* which generates the code for the control variables (attributes of the model), erases the auxiliary attributes (*current* and *visited*) from the entities, and closes the output file.

The advantage of using a graph grammar to generate the textual code is that it is done in a graphical, high-level fashion, and the user does not need any knowledge of compilers nor of the AToM^3 Kernel internals.

The result of the automatic code generation for the model of a telephone exchange partially visible in Figure 3 is shown in Figure 5.

7 Related work

Apart from AToM^3, other visual meta-modelling tools exist. Examples are DOME [8], Multigraph [24], MetaEdit+ [18] and KOGGE [10]. Some of these allow one to express formalism semantics by means of some kind of textual language. For example, KOGGE uses a language similar to Modula-2. Our approach is quite different, as we express such semantics by means of graph grammars. We believe graph grammars are a natural and general way to express graph manipulation. A rationale for using graph grammars in our approach was given in section 3. Also, none of the tools consider the possibility to transform models between different formalisms.

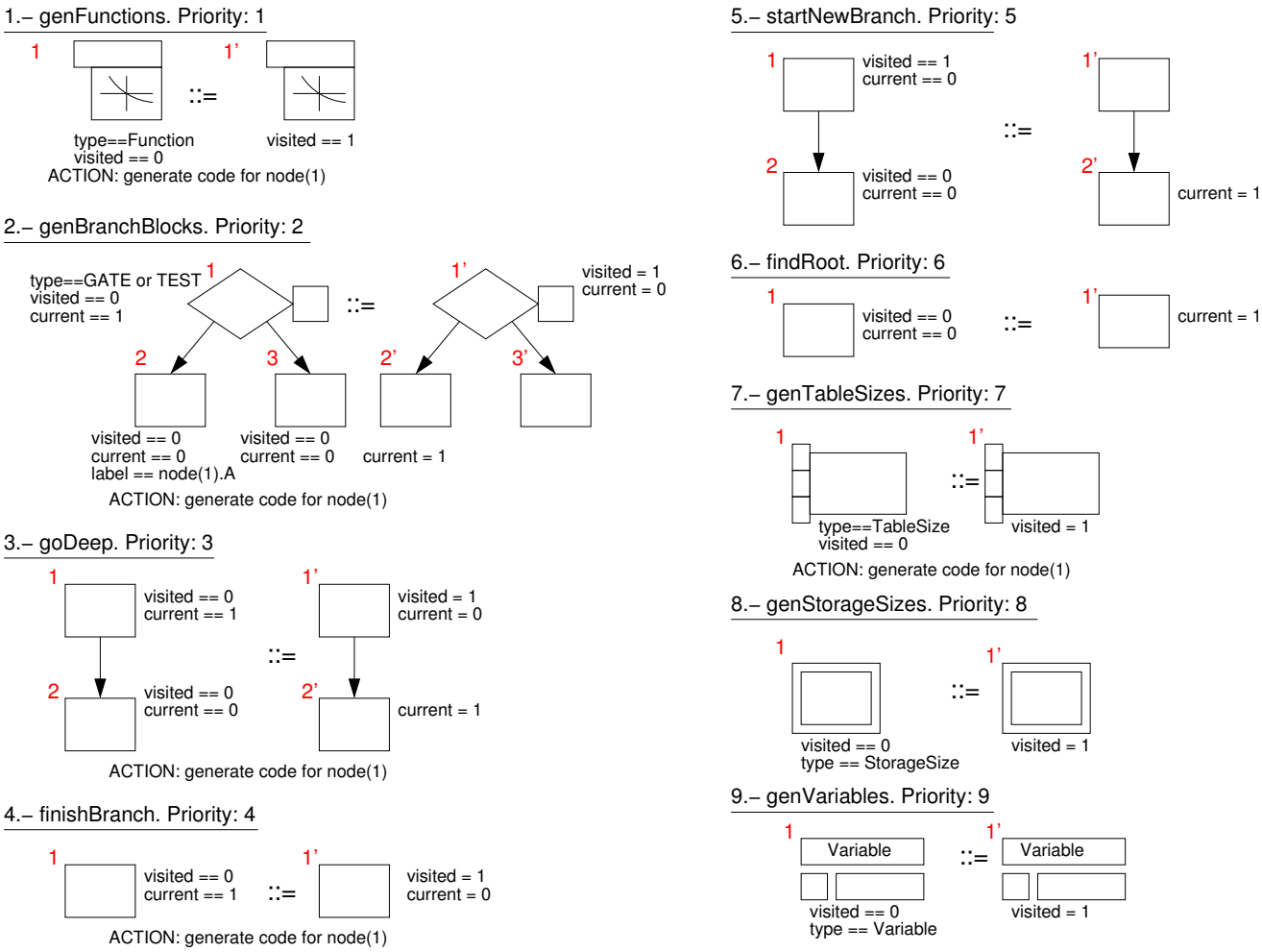

Figure 4: A graph grammar to generate code for a simulator from a GPSS model.

Systems and languages exist for graph grammar manipulation. Examples are PROGRES [21], GRACE [13], AGG [2], etc. All lack of a Meta-Modelling layer.

In 1997, the OMG [20] proposed a Meta-Data standard, called the Meta-Object Facility (MOF) [19]. In this approach, meta-models are described using UML and are stored in a standard format. From these meta-models, by means of XMI, it is possible to automatically obtain DTDs and XML documents. We are invesitgating to make the way we store (meta-)models compatible with the MOF. Of particular interest is the standard representation of model transformations (in the form of graph grammar models), as they would allow the translation of data between different DTDs in a meaningful and automated fashion.

Our approach is original in the sense that we take the advantages of *Meta-Modelling* to avoid explicit programming of customized tools and combine them with those of *graph transformation* systems to express model behaviour, formalism transformation. Our main contribution is thus in the field of *multi-paradigm modelling* [25] as we provide a general means to manipulate and transform models between different formalisms.

8 Conclusions and future work

In this paper we have discussed the advantages of meta-modelling to obtain customized modelling environments in an automatic way. Building the application from scratch is no longer necessary, we only have to specify –in a graphical way– the meta-model of the formalism we are interested in. The processing of models in the described formalism can be expressed at the meta-level by means of graph grammars. Our approach is also highly applicable if we want to construct a slight variation of some formalism. In this case, we only need to specify the meta-model for the new formalism and a transformation into a "known" formalism (one that already has a transformation to generate model code to be used as input for a simulator, for example).

We have presented AToM^3, a tool which implements the concepts presented before, and demonstrated its usefulness by generating a modelling tool for GPSS models able to produce textual code for further simulation. As we also

```
      SIMULATE
 1    FUNCTION   RN1,C24
0.0,0.0/0.1,0.104/0.2,0.222/0.3,0.355/0.4,0.509/0.5,0.69
0.6,0.915/0.7,1.2/0.75,1.38/0.8,1.6/0.84,1.83/0.88,2.12
0.9,2.13/0.92,2.52/0.94,2.81/0.95,2.99/0.96,3.2/0.97,3.5
0.98,3.9/0.99,4.6/0.995,5.3/0.998,6.2/0.999,7.0/0.9997,8.0
*
 L1    GENERATE   12,FN1
 L3    TEST G     V2,2,OUT
 L4    ASSIGN     1,V1,H
 L5    GATE LR    PH1,L4
 L6    ASSIGN     2,V1,H
 L7    TEST NE    P1,P2,L6
 L0    LOGIC R    PH1
 L8    TRANSFER   BOTH,L9,L11
 L9    LOGIC R    PH1
 L10   TERMINATE  1
 OUT   TERMINATE  0
 L11   ENTER      LNKS
 L12   GATE LR    PH2,L13
 L16   LOGIC S    PH2
 L17   ADVANCE    120,FN1
 L18   LOGIC R    PH1
 L19   LOGIC R    PH2
 L20   LEAVE      LNKS
 L21   TERMINATE  1
 L13   LOGIC R    PH1
 L14   LEAVE      LNKS
 L15   TERMINATE  1
 LNKS  STORAGE    10
 1     VARIABLE   XH1*RN1/1000+1
 2     VARIABLE   XH1-2*S$LNKS
       START      1000
       END
```

Figure 5: Synthesized GPSS code

generate a model of part of the user interface, the tool can be customized (in a graphical way) very easily. Thanks to the combined approach of Meta-Modelling and Graph Transformations, we were able to build the complete environment in one day. This environment was subsequently used in a Modelling and Simulation course at McGill University to build GPSS models, generate the GPSS textual programs and run these programs in the HGPSS Simulator [6].

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

In the future, we will extend the GPSS meta-model to take advantage of the hierarchical possibilities of HGPSS. We also plan to extend AToM^3 in other ways:

- Describing another meta-meta-model in terms of the current one (the Entity-Relationship meta-meta-model). In particular, we plan to describe UML class diagrams. For this purpose, relationships between classes such as *inheritance* must be described. Thanks to our meta-modelling approach, we are able to describe different subclassing semantics and their relationship with subtyping [1]. Furthermore, as the semantics of inheritance will be described at the meta-level, code can be generated in non-object-oriented languages. A similar approach (meta-modelling + bootstrapping) to model UML class diagrams is also proposed by Harel in [16].
- Allow collaborative modelling: for this purpose, we are putting the APIs for constructing graphical interfaces in Java (Swing) and Python (Tkinter) at the same level. These developments, together with the possibility of using Python on top of the Java Virtual Machine (*e.g.,* by means of Jython [17]), will allow us to make AToM^3 accessible through a web browser in applet form. 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 the OMG's MOF in combination with XML.

Acknowledgments

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] M. Abadi and L. Cardelli 1996. *A Theory of Objects*. Monographs in Computer Science. Springer.

[2] AGG Home page: http://tfs.cs.tu-berlin.de/agg/

[3] A.V. Aho, R. Sethi, and J.D. Ullman 1986. Compilers, principles, techniques and tools. Chapter 6, Type Checking. Addison-Wesley.

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

[5] D. Blonstein, H. Fahmy, and A. Grbavec 1996. *Issues in the Practical Use of Graph Rewriting*. Lecture Notes in Computer Science, Vol. 1073, Springer-Verlag, pp.38-55.

[6] F. Claeys, H. Vangheluwe, and G.C. Vansteenkiste 1995. HGPSS: A hierarchical extension to GPSS. In M. Snoreck, M. Sujansky, and A. Verbraeck, editors, *Procedeedings of the 1995 European Simulation Multiconference (ESM)*. Society for Computer Simulation International (SCS), June 1995, pp.138-142.

[7] J. de Lara and H. Vangheluwe 2002. AToM^3: A tool for multi-formalism and meta-modelling. In European Joint Conference on Theory And Practice of Software

(ETAPS), Fundamental Approaches to Software Engineering (FASE), Lecture Notes in Computer Science 2306, pages 174 - 188. Springer-Verlag, April 2002. Grenoble, France.

[8] DOME guide. http://www.htc.honeywell.com/dome/, Honeywell Technology Center. Honeywell, 1999, version 5.2.1

[9] H. Dorr 1995. *Efficient Graph Rewriting and its implementation*. Lecture Notes in Computer Science 922. Springer-Verlag.

[10] J. Ebert, R. Sttenbach, and I. Uhe 1997. Meta-CASE in Practice: a Case for KOGGE. Advanced Information Systems Engineering, Proceedings of the 9th International Conference, CAiSE'97 LNCS 1250, S. 203-216, Berlin, 1997. See KOGGE home page at: http://www.uni-koblenz.de/ ist/kogge.en.html

[11] G. Gordon 1996. *System Simulation*, Prentice Hall. Second Edition.

[12] I. Ståhl 2001. *GPSS - 40 Years of Development*. Proceedings of the 2001 Winter Simulation Conference. Washington, DC.

[13] GRACE Home page: http://www.informatik.uni-bremen.de/theorie/GRACEland/GRACEland.html

[14] J. Gray, T. Bapty, and S. Neema 2000. *Aspectifying Constraints in Model-Integrated Computing*, OOPSLA 2000: Workshop on Advanced Separation of Concerns, Minneapolis, MN, October, 2000.

[15] D. Harel 1988. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[16] D. Harel and B. Rumpe 2000. *Modeling languages: Syntax, Semantics and All That Stuff. Part I: The Basic Stuff* Technical Report MCS00-16 in the Belfer Institute of Mathematics and Computer Science.

[17] Jython Home Page: *http://www.jython.org*

[18] MetaCase Home Page: http://www.MetaCase.com/

[19] Meta-Object Facility, from the OMG: *http://www.omg.org/cwm*

[20] OMG Home Page: *http://www.omg.org*

[21] PROGRES home page: *http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html*

[22] Python home page: *http://www.python.org*

[23] T. Schriber 1974. *Simulation Using GPSS*. Wiley.

[24] J. Sztipanovits, et al. 1995. MULTIGRAPH: An architecture for model-integrated computing. In ICECCS'95, pp. 361-368, Ft. Lauderdale, Florida, Nov. 1995.

[25] H. Vangheluwe 2000. *DEVS as a common denominator for multi-formalism hybrid systems modelling.* In Andras Varga, editor, *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134. IEEE Computer Society Press, September 2000. Anchorage, Alaska.