Using AToM^3 as a Meta-CASE Tool

Juan de Lara[1,2]
[1]*ETS Informática, Universidad Autónoma de Madrid*
Ctra. Cantoblanco km.15, Madrid, Spain
Juan.Lara@ii.uam.es

Hans Vangheluwe[2]
[2]*School of Computer Science, McGill University*
3480 University Street, Montreal, Canada
hv@cs.mcgill.ca

Keywords: Meta-Modelling, Multi-Formalism Modelling, Graph Grammars, Meta-CASE tools, Visual Languages.

Abstract:

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 – possibly large numbers of – components and aspects whose structure as well as behaviour cannot be described in a single formalism. Examples of commonly used modelling formalisms are Differential-Algebraic Equations (DAEs), Bond Graphs, Petri Nets, DEVS, Entity-Relationship (ER) diagrams, and Structure Charts (SC).

Meta-CASE generators are tools capable of generating customized CASE tools. They are useful when some minor variation of a formalism is needed for which building a complete tool from scratch would be too expensive. Meta-CASE tools solve this by allowing a graphical, high-level description of the CASE tool to be generated. Examples of such tools are KOGGE (Ebert et al., 1997) and MetaEdit+ (Kelly et al., 1996).

In analogy with string grammars, graph grammars (Dorr, 1995) 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 (called host graph), this subgraph is replaced by the right-hand side of the rule. A rewriting system iteratively applies matching rules in the grammar to the graph, until no more rules are applicable.

In this paper, we present AToM^3 (A Tool for Multi-Formalism and Meta-Modelling). This tool has a meta-modelling layer in which different graphical formalisms are modelled. From the meta-specification (in the ER formalism extended with constraints), AToM^3 generates a tool to process models described in the specified formalism. Models are internally represented using *Abstract Syntax Graphs*. As a consequence, manipulations of models can be expressed as graph-grammars models. The latter are also specified graphically in AToM^3. Examples of model manipulations are: optimization of models, code synthesis, transformation of a model into another (behaviourally-equivalent) model expressed in a different formalism and simulation. Although graph grammars have been used in very diverse areas such as graphical editors, code optimization and computer architecture (Ehrig et al., 1991), to our knowledge, they have never been applied to formalism transformations.

Up to now we have used AToM^3 to model simulation formalisms and to transform models between such formalisms. In this paper, we demonstrate the power of combining meta-modelling and graph transformations by using the tool in a different application domain, namely as a meta-CASE tool.

2 META-MODELLING

Apart from a large number of components, one of the characteristics of complex systems is the diversity of these 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

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.

achieve the required flexibility for a modelling language that supports many formalisms and modelling paradigms is to model the modelling language itself (DOME, 1999) (Sztipanovits et al., 1995). 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 2 depicts the levels considered in our meta-modelling approach.

Formalisms such as ER or UML class diagrams (Precise UML group, 2002) are often used for meta-modelling. To be able to fully specify modelling formalisms, the meta-level formalism must 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 ER 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 (Sztipanovits et al., 1995), including AToM^3 use the Object Constraint Language OCL (OMG, 2002) used in the UML.

Figure 1 depicts the structure we propose for a meta-modelling environment. For example, in AToM^3, the Meta-Meta Model contains a description of ER, initially hand-coded. As the ER formalism can be described in an ER model, the environment was subsequently bootstrapped, although other meta-formalisms (such as the UML) can also be described. Details about the AToM^3 internals will be described in section 4

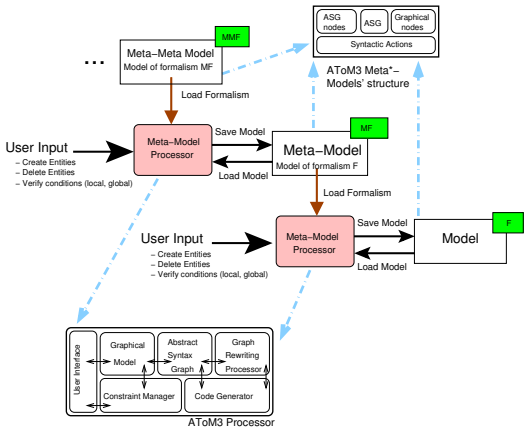

Figure 1: Proposed working scheme for a meta-modelling environment.

3 GRAPH GRAMMARS

In our approach, we use graph grammars to express model manipulation. As stated in the introduction, they are composed of rules, each mapping a graph on the left-hand side (LHS) to a graph on the right-hand side (RHS). A graph grammar is applied to an input graph (called host graph) in order to perform a transformation. When a match is found between the LHS of a rule and a part of the host graph, this subgraph is replaced by the RHS 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 (Dorr, 1995). Some approaches also offer control flow specifications. In our tool, rules are ordered based on a user-assigned priority.

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) (Blonstein et al., 1996):

- It is an abstract, declarative, high level representation. This enables exchange, re-use, and symbolic analysis of the transformation model.
- 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, sub-

graph 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.

4 AToM^3

AToM^3 is a tool written in Python (Python, 2002) which implements the above ideas. Its architecture was shown on figure 1.

The main component of AToM^3 is the *Processor*, 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. The first kind of models allow constructing valid models in a certain formalism, the latter are used to describe the formalisms themselves. In AToM^3 models at any meta-level are treated in the same way.

The ER 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 (name and type). We will refer to this as the semantic information. For example, to define the Data Flow Diagram formalism (DFD), it is necessary to define *Data Flows*, *Processes*, *External Entities* and *Data Stores*. Furthermore, we need to specify attributes for each of these entities. For example, a *Process* has a *Name* (type: String), an *ID number* (type: Integer), a *textual description* (type: String), etc.

In AToM^3 a distinction is made between 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. This is useful if we are trying to model another meta-meta-model, such as UML class diagrams, or when the ER meta-meta-model was bootstraped. Both types of attributes may contain data or code for pre- and post-conditions. Only meta-meta-level entities are provided with generative attributes.

The above specification is used by the AToM^3 *Processor* to generate some Python files, which, when loaded by the *Processor*, allows manipulation of models in the defined formalism.

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 DFDs, we can choose to represent *Processes* as ovals with the *name* and the *ID number* inside. That is, we can specify how some semantic attributes are displayed graphically. Constraints can also be associated with the graphical entities. Each graphical form, part of the graphical entity, can be referenced through an automatically generated name which has methods to change its color, hide it, etc.

AToM^3 has a number of basic types, such as Strings, Integers, and Floats. Each type in AToM^3 has associated a Python class, which is responsible for creating a widget to edit its value, checking the validity of its value, making the variable persistent, etc.

It is also possible to define composite types. Following the tool's philosophy, composite types are models. Which models are valid is described using a *Types* meta-model. Thus, types are represented as directed graphs, as proposed in (Aho et al., 1996). Type models have a *Root* node, which is labeled with the type's name. From the *Root* node, *Operator* nodes can be connected. The *Operator* type can be either *Product* (to build tuples) or *Union*. *Operator* nodes can be connected to other *Operator* nodes, to the *Root* node (that is, we allow recursive types) or to *SubType* nodes. The *SubType* node type can be any valid type (basic or composite) in the current AToM^3 session. *SubType* nodes cannot have any outgoing connection, thus, these kind of nodes are the graph leaves. *SubType* nodes may have associated a constraint to restrict the type value. This is useful for example if we try to define subranges of a type. A tool for processing types was generated automatically from this meta-description and then incorporated into the AToM^3 *Processor*. Consequently, types may be loaded, saved and manipulated as any other model. The meta-level has been constrained to avoid infinite recursion in the type definition.

Some graph grammars have been defined to process the type models. For example, we have implemented a graph-grammar to generate the Python class associated with the type.

For the implementation of the Graph Rewriting Module, we have used an improvement of the algorithm given in (Dorr, 1995), in which we allow non-connected graphs in LHSs in rules. 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 model optimization. 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), if the user can choose one of the matching subgraphs interactivelly, or the 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).

5 AN EXAMPLE: GENERATING TOOLS FOR STRUCTURED ANALYSIS AND DESIGN

5.1 Defining the meta-models

In this section, we will use AToM^3 to describe two meta-models for structured analysis (DFD) and design (SC).

We need three entities to define the DFD formalism: *Processes*, *External entities*, and *Data Stores* and a *DataFlow* relationship. In addition, we need to specify some constraints to prohibit drawing invalid models, namely:

- Connections of two *External Entities* by means of a *DataFlow* is not allowed. We have expressed this as a local constraint on the entity *DataFlow*, which will be evaluated when trying to connect it.
- *Data Stores* can only be read or written by *Processes*. This will also be expressed as a local constraint on *DataFlows*, evaluated when trying to connect it.
- All *DataFlow* names must be unique. This can only be implemented as a global constraint. As it may not be desireable to maintain this restriction during the whole modelling process, it will only be evaluated when trying to save the model. This ensures that all the saved models are valid.
- All *Process*, *External Entity* and *Data Store* names must also be unique. This is implemented in a similar way as above. *Processes* ID numbers must be unique too.

Figure 2 shows AToM^3 loaded with the ER Meta-Meta-Model, which is used to describe the DFD formalism. The dialog to edit the *"Process"* entity is opened, and its graphical representation is being edited. In Figure 3 AToM^3 is loaded with the generated DFD meta-model. It is noted that the entities that can be created using this meta-model are different from the ones that can be created with the ER meta-meta-model.

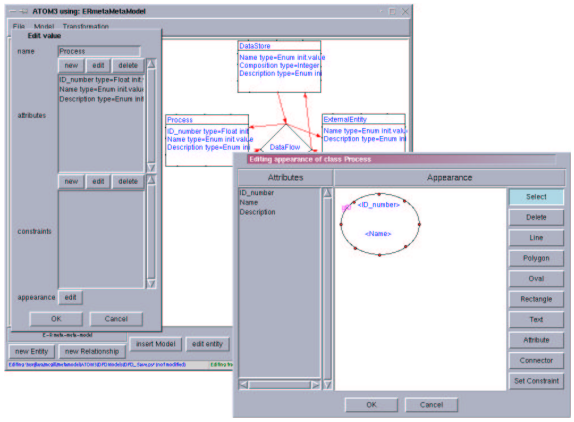

Figure 2: Describing DFDs with AToM^3.

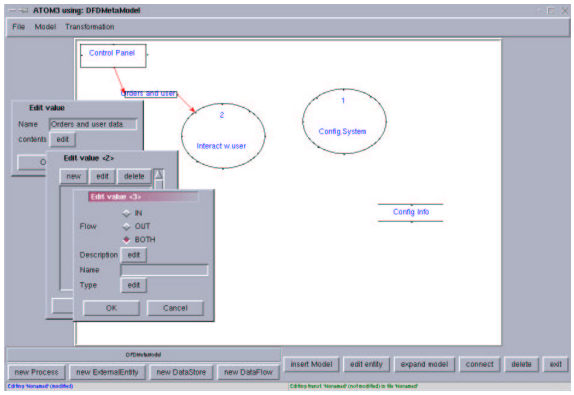

Figure 3: Generated DFD Tool.

The *DataFlow* relationship has been provided with some attributes: a *Name* and a list of *Data* elements. *Data* is a 4-element tuple (a composite type) containing: a *Name* (string), a *Type* (an enumerated type containing the valid types), a *Flow* (an enumarated type indicating the flow direction of the data, IN, OUT or BOTH) and a *Description* (string). In Figure 3, a *DataFlow* is being edited, and a *Data* element is being added to the list. The dialog window to edit these *Data* elements has been generated from the type definition with a graph-grammar.

The SC meta-model is simpler as we only have *Modules* and *Data Flows*. The latter are very similar to the ones in the previous meta-model, the former are composed of a *name* and a *description*.

For this meta-model, there are also some constraints:

- The number of incoming connections of any module entity must be strictly positive.
- The number of *Data Flow* elements without incoming connections must be equal to one. This means that a unique root node is present in the model.
- *Module* and *Data Flow* names must be unique.

The first condition can be implemented as a local condition. The other two need information about the whole model, so they should be global.

5.2 Defining model transformations

In this section, we will present some graph grammars we have defined to manipulate DFD and SC models. The first one is used to transform DFD into SC models, the second one to optimize the resulting SC models. It must be noted, that there are suggestions in the literature on how to transform a DFD model into a SC model (see for example (Pressmanm, 1997)), but to our knowledge, this is the first time that these suggestions have been implemented as a graph grammar for the purpose of automatic formalism transformation. Of course, these are suggestions and further manual manipulation of the resulting models may be necessary.

5.2.1 Transforming DFDs into SCs

In general, two strategies can be used when transforming a DFD into an SC: transformation analysis and transaction analysis. For brievity, we will only deal with the first one. Transformation analysis is applied if there is a zone in the DFD model with transformation characteristics, which are:

- One or more processes that capture data and convert it to some internal format. This format facilitates the manipulation of the data.
- One or more processes dealing with the manipulation of the data.
- One or more processes dealing with the transformation of the data into another format suitable for output.

To implement this graph grammar, one field has been added to the entity *Process* in the DFD meta-model to indicate if the process deals with input, output, transaction or transformation. Currently, this field must be filled by the user, although in the future we are planning to build a graph grammar to identify transaction nodes. This field has also been added to the *Module* entity in the SC meta-model, but is filled by the graph grammar.

Figure 4 shows a part of the graph grammar used to transform a DFD model into a SC model. For brevity, only some of the rules involved in the transformation analysis are presented. Rules dealing with the transaction analysis are also part of this graph grammar.

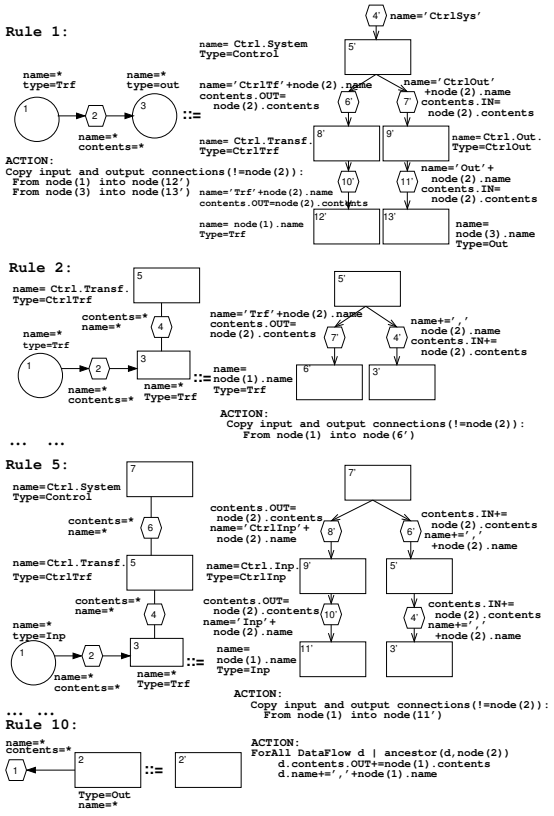

Figure 4: Some rules of the graph-grammar to transform a DFD model into an SC model.

In this graph grammar, entities are labeled with numbers. RHS node labels are also decorated with a prime, to distinguish them from LHS ones. If two nodes in a LHS and a RHS have the same number, the node must not disappear when the rule is executed. If a number appears in a LHS but not in a RHS, the node must be removed when applying the rule. If a number appears in a RHS but not in a LHS, the node must be created if the rule is applied. If a node in a RHS is empty, their attributes will be left untouched, otherwise their value is specified. For clarity, the *contents* attribute of *DataFlows* is considered to have two lists (*IN* and *OUT*) which contain the input and output parameters to the module.

These rules will be tried in order by the graph rewriting system. Rule number 1 is the first one to be applied when a transformation zone is detected in the DFD model. It starts building the transformation/output zone of the SC model. The second and third rules add transformation processes to the SC model, the fourth, fifth and sixth rules add input processes to the SC model, rule seven adds output processes to the model, rules eight, nine and ten elim-

inate DFD data flows between SC modules, adding this data information properly to the corresponding SC data flows. These three last rules incorporate some actions to propagate the information found in the *DataFlow* that is being deleted up in the modules hierarchy.

Figure 5 shows AToM^3 after the execution of rule 5 on a DFD model.

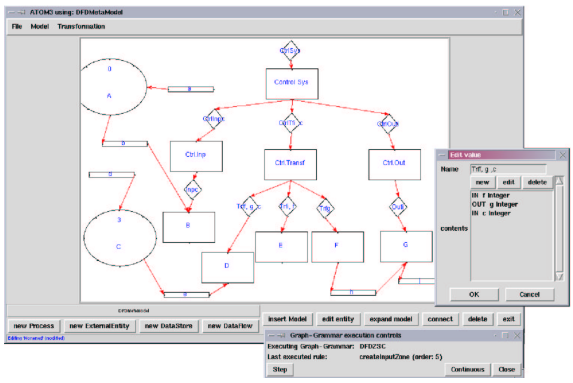

Figure 5: AToM^3 after execution of rule 5.

As it can be observed, during the application of the graph-grammar, the model is a blend of a DFD model and a SC model. At the end of the process however, the model is entirely in the SC formalism. This implies that during transformation, the DFD and SC constraints check must be turned off, as stated in section 4.

5.2.2 'Optimizing' SC models

We have implemented rules for simplification of SC models in a different graph-grammar. This graph grammar "optimizes" SC models, and hence, does not perform formalism change. In our example, it could be useful to execute this optimizing graph-grammar after applying the one described before to a DFD (AToM^3 allows applying sequences of graph-grammars). Some of the rules of this optimizing graph-grammar eliminate a control module if it has only one "child" and others provide a module with two "children" modules if it is considered complex.

Once the model is in the SC formalism, it is easy to define other graph grammars, for example to generate code skeletons. In essence, the graph grammar has to go through the modules hierarchy, and for each module, take its input *DataFlow* to generate the function prototype, and for each *DataFlow* connected as output, generate a function call.

6 Related work

A similar approach is ViewPoint Oriented Software Development (Finkelstein, 1990). Some of the concepts introduced by the authors have a clear counterpart in our aproach (for example, *ViewPoint templates* are equivalent to meta-models, etc). They also introduce the relationships between ViewPoints, which are similar to our coupling of models and graph transformations.

Although this approach has some characteristics that our approach lacks (such as the *work plan axioms*), our use of graph transformations allows to express model's behaviour and formalism's semantics. These graph transformations allow us to transform models between formalisms, optimize models, or describe basic simulators. For example, we have implemented a simulator for block diagrams using graph grammars (AToM3, 2002). Another advantage of our approach, is that we use meta-modelling, in this way we don't need different tools to process different formalisms (ViewPoints), as we can model them at the meta-level.

Other approaches to interconnecting formalisms are Category Theory (Fiadeiro, 1995), in which formalisms are cast as categories and their relationships as functors. See also (Zave and Jackson, 1993) and (Niskier at al., 1989) for other approaches.

There are other visual tools to describe formalisms using meta-modelling, among them DOME (DOME, 1999), Multigraph (Sztipanovits et al., 1995), MetaEdit+ (Kelly et al., 1996) or KOGGE (Ebert et al., 1997). Some of them allow to express formalism semantics by means of a textual language (KOGGE for example uses a Modula-2-like language). Our approach is quite different. We express semantics by means of graph grammar models. We believe that graph grammars are a natural, declarative, and general way to express transformations. As graph grammars are highly amenable to graphical representation, they are superior to a purely textual language. Some of the rationale for using graph grammars in our approach has been show in section 3. Also, none of the tools consider the possibility to transform models between different formalisms.

On the other hand, there are some systems and languages for graph-grammar manipulation, such as PROGRES (PROGRES, 2002), GRACE (GRACE, 2002) and AGG (AGG, 2002). None of these have a Meta-Modelling layer. Work on using graph-grammars to specify software architecture transformations can be found at (Fahmy et al., 2000).

Our approach is original in the sense that we take the advantages of Meta-Modelling (e.g., to avoid explicit programming of customized tools) and combine them with those of graph transformation systems (e.g., to express model behaviour, formalism transfor-

mation). Our main contribution is thus in the field of multi-paradigm modelling (Vangheluwe, 2000) as we provide a general means to transform models between different formalisms and to manipulate them.

7 Conclusions and future work

In this paper, we have presented AToM^3, a tool for multi-formalism meta-modelling. The meta-modelling layer allows a high-level description of models. Using this meta-information, AToM^3 can generate automatically a tool to process these models. Manipulations of models can be expressed as graph grammars, at the meta-level. Some of these manipulations are the behaviour-preserving transformations of models between formalisms, optimization, code generation and simulation.

Particularlly, in this paper we have shown the use of AToM^3 to generate a structured analysis and design CASE tool. A graph grammar has been defined to transform DFD models into SC models. Our approach allows the creation of new tools (by meta-modelling them with AToM^3) and connecting them by defining appropriate graph-grammars to transform models between tools. For example, it is possible to define a graph grammar to extract information about the module parameters in the SC model and use it to produce test cases.

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. This highly reduces the effort needed to build such a tool. For example, the tools described in this paper have been built in a few hours. Our approach is also highly applicable if we want to work with a slight variation of some formalism, where we only have to slightly modify the formalism's meta-model. We may also specify the meta-model for a new formalism and a tranformation into a "known" formalism (one that already has the appropriate transformation available, such as a simulator). AToM^3 is available at (AToM3, 2002).

In the future, we will explore software process modelling (Koskinen and Marttiin, 1997). A process may be composed of different tasks, each one implying the use of a different meta-model to obtain the desired product. Also, it could be useful to let the degree of specificity (i.e. the constraints that the model must satisfy to be valid) of a modelling formalism change during the modelling process. It is envisioned that this evolution of the formalism during the modelling life-cycle will eventually be specified using a variable-structure meta-model (such as a DFA with ER states).

In 1997, the OMG (OMG, 2002) proposed a Meta-Data standard, called the Meta-Object Facility (MOF) (MOF, 2002). 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 consider the possibility to make compatible with MOF the way we store (meta-)models. 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.

We are also planning to extend the tool in several ways:

- Describing another meta-meta-model in terms of the current one (the ER meta-meta-model) is possible. In particular, we will describe UML class diagrams. For this purpose, relationships between classes such as inheritance need to be described. Due to our meta-modelling approach, we will be able to describe different subclassing semantics and its relationships with subtyping (Abadi, 1996).
- We should explore more in detail the implications of hierarchical modelling, and their relationship with graph transformations. Although our tool allows for hierarchical modelling, further study is needed on how to apply graph grammars on different hierarchical levels of the model. For this purpose, the replacement of the basic internal data structure for representing models (graphs) by the more expressive HiGraphs (Harel, 1988) is under consideration. HiGraphs are more suitable to express and visualize hierarchies, they add the concept of orthogonality, and connections use hyperedges.
- We will extend 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 OMG's MOF combined with XML.

ACKNOWLEDGMENT

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

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

AGG Home page: *http://tfs.cs.tu-berlin.de/agg/*

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

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

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

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

Dorr, H. 1995. *Efficient Graph Rewriting and its implementation*. Lecture Notes in Computer Science, 922. Springer.

Ebert, J., Sttenbach, R., Uhe, I. 1997 *MetaCASE in Practice: a Case for KOGGE* In A. Olive, J. A. Pastor: Advanced Information Systems Engineering, Proceedings of CAiSE'97, Barcelona. LNCS 1250. pp.:203-216, Springer. See KOGGE home page at: *http://www.uni-koblenz.de/ ist/kogge.en.html*

Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) 1991. *Graph Grammars and their application to Computer Science: 4th International Workshop, Bremen, Germany, March 5-9, 1990, Proceedings*. LNCS, 532. Springer.

Fahmy, H., Holt, R.C. 2000. *Using Graph Rewriting to Specify Software Architectural Transformations*, Proceedings of 15th IEEE Conference on Automated Software Engineering, Grenoble, France.

Fiadeiro, J.L., Maibaum, T. 1995. *Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality* Proc.3rd Symposium on the Fundations of Software Engineering, G.E.Kaiser(ed),pp.: 72-80, ACM Press.

Finkelstein, A., Kramer, J., Goedickie, M. *ViewPoint Oriented Software Development* Proc, of the Third Int. Workshop on Software Engineering and its Applications, Tolouse, December 1990.

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

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

Kelly, S., Lyytinen, K., Rossi, M. *MetaEdit+: A fully configurable Multi-User and Multi-Tool CASE and CAME Environment* In Constantopoulos, P., Mylopoulos, J., Vassiliou, Y: Advanced Information System Engineering; LNCS 1080. Berlin: Springer 1996. See MetaEdit+ Home page at: *http://www.MetaCase.com/*

Koskinen, M.; Marttiin, P. 1997. *Process support in MetaCASE: implementing the conceptual basis for enactable process models in MetaEdit+*. Eighth Conference on Software Engineering Environments, pp.: 110-122.

Meta-Modelling Facility, from the precise UML group: http://www.cs.york.ac.uk/puml/mmf/index.html

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

Niskier, C., Maibaum, T., Schwabe, D. 1989 *A pluralistic Knowledge Based Approach to Software Specification* 2nd European Software Engineering Conference, LNCS 387, Springer Verlag 1989, pp.:411-423

OMG Home Page: *http://www.omg.org*

Pressman, R.S. 1997. *Software Engineering: A practitioner's approach*. McGraw-Hill.

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

Python home page: *http://www.python.org*

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

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

Zave, P., Jackson, M. 1993. *Conjunction as Composition* ACM Transactions on Software Engineering and Methodology 2(4), 1993, 371-411.