Towards the Uniform Manipulation of Visual and Textual Languages in AToM^3

Juan de Lara and Esther Guerra

Escuela Politécnica Superior
Ingeniería Informática
Universidad Autónoma de Madrid
{Juan.Lara|Esther.Guerra_Sanchez}@ii.uam.es

Abstract. This paper presents the approach taken in the *multi-paradigm* tool AToM^3 for the integration of textual and visual languages in a uniform framework. The tool is used for the modelling, analysis and simulation of complex (physical or software) systems, where each system component may have to be described using a different formalism. The different visual or textual formalisms can be described in the form of meta-models using graphical, high-level notations such as Entity Relationship or UML class diagrams. From these descriptions, AToM^3 is able to generate a customized modelling tool for the specified formalism. Models at any meta-level are stored as attributed, typed graphs and thus can be manipulated (simulated, transformed, optimized, etc.) by attributed graph grammars.
In the case of a textual notation, from the meta-model description a front-end parser is semi-automatically generated that transforms the textual models into *abstract syntax graphs* (instances of the meta-model), and thus can be manipulated in a uniform way with the other visual notations. To illustrate these concepts, we present an example in which we define a meta-model for Computational Tree Logic and generate visual and textual parsers for the formalism.

1 Introduction

Complex (physical or logical) systems are made of different components or views, each one of them may have to be described using a different notation, due to their different nature. Thus, for the modelling of such a system, one would like to use the most appropriate formalism for each component. Some of these formalisms may be graphical, while others may be textual.

Visual models of software systems are ever more frequently not just a mere design or analysis documentation, but play a more active role in the development process. In ideal automated development environments, software is actually built by means of high-level, visual notations. This approach can be seen as a further step in the search for higher-levels of productivity and quality by raising the abstraction level of languages. This, in fact, is an evolution we are witnessing for as long as computers were invented more than half a century ago: from bits to assembler to procedural programming languages to object-oriented languages up to visual notations. Using higher abstraction level notations, programs become more compact and easier to understand, write and

maintain. In this way, developers deal with less (*accidental*) details about the system they are building and concentrate on describing its *essential* properties [2]. Although highly expressive, for convenience, visual notations are often combined with textual languages. This is the case of the different UML diagrams and the textual constraint language OCL [20].

Multi-Paradigm Modelling [19] proposes the combination of *meta-modelling*, *multi-formalism* and *multiple levels of abstraction* for the modelling and analysis of complex systems. By means of meta-modelling one can reduce the effort of obtaining a customized modelling tool for a certain formalism. Meta-modelling consists on modelling the formalisms themselves (its syntax) by means of high level languages such as Entity Relationship or UML class diagrams. From these meta-models, a meta-modelling tool is able to produce a customized modelling tool for the specified formalism (see for example [13] [11] [6]).

Multi-formalism modelling allows the description of complex systems using different formalisms. For the analysis of such a system, each component or view is translated into a common formalism, where the properties of interest can be investigated. The Formalism Transformation Graph [19] (FTG) may help in finding such a common formalism. It is shown in Figure 1 and depicts in the form of a graph a small portion of the *"formalism space"*. Formalisms are shown as nodes in the graph. Some of them – those shown enclosed in rounded rectangles – are graphical (such as automata, Petri nets, Statecharts, etc.), while some others are inherently textual (for example, formalisms based on equations). Thus, for the specification and analysis of a multi-formalism system it is necessary to integrate textual and graphical modelling tools. It is also desirable to have an easy and uniform way to specify and obtain both kinds of modelling tools.

The arrows between the nodes in the FTG denote behaviour-preserving transformations, but they may indeed lead to a loss of information. For example, when transforming from Petri nets into Automata (that is, when calculating the *coverability graph*), the information about the exact number of tokens at each state (marking) may be lost if the net is unbounded. Nonetheless, by performing the transformation we are able to solve questions that were harder or impossible to answer in the source formalism. The vertical, dashed arrows denote the existence of a simulator for the formalism. This indeed can be seen as a special case of iterative transformation into the *"execution traces"* formalism. The dotted loop-arrows denote the existence of optimization transformations (to reduce complexity, improve performance, etc.) for the formalism.

Our approach is describing formalisms (of any kind) in the FTG by means of meta-modelling. As models at any meta-level can be stored as attributed, typed graphs, they can be manipulated using graph grammars [15]. Thus we model the arrows (of any kind) in the FTG as graph transformations.

These ideas have been implemented in the *multi-paradigm* tool AToM^3 [6], which was built in collaboration with McGill University in Montreal. This paper describes the first steps towards the uniform processing and integration of textual and graphical notations, and its implementation in AToM^3. The approach consists on describing textual formalisms in the same way as graphical notations, through a meta-model. Instead of assigning a visual concrete syntax to the notation, we transform the meta-model (using a graph grammar) into a *string grammar*. This grammar is input into an automatic

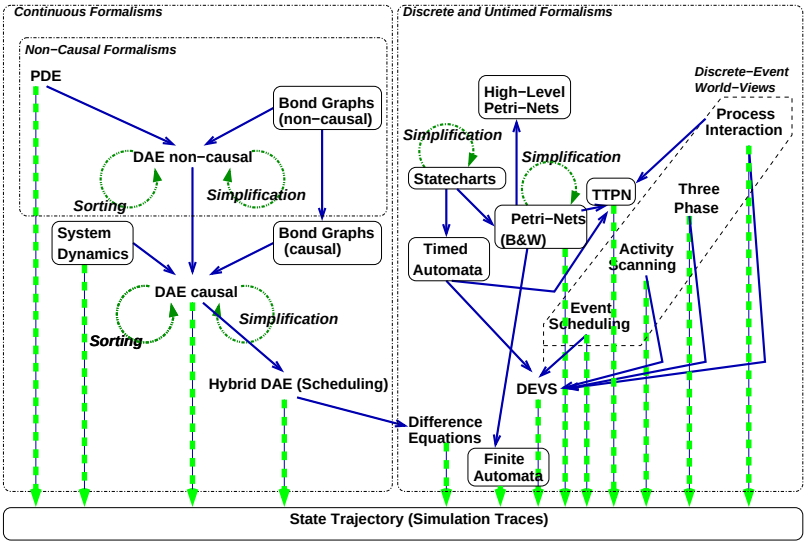

Fig. 1. Formalism Transformation Graph (adapted from [19]).

parser generator (we have used TRAP [9]). The generated parser is then able to process textual programs and generate *abstract syntax graphs* that are valid instances of the original meta-model. Integrating the generated textual parser into AToM^3 makes it possible to process textual as well as graphical notations in a uniform way.

The problem we are dealing with in this paper is highly relevant for the UML community, which faces a similar problem: integrating the different UML diagrams with the textual constraint language OCL [20]. The OMG has taken a similar approach to the one we are trying to automate (although of course, for much simpler languages than OCL). They defined the abstract syntax of OCL by means of a (MOF compliant) meta-model. The (textual) concrete syntax is defined by an attributed (EBNF) string grammar. Productions build a valid instance of the meta-model by means of synthesized attributes (see section 5).

This paper is organized as follows: section 2 gives a brief, informal overview of graph grammars. Section 3 introduces the AToM^3 tool. Section 4 presents a meta-model for Computational Tree Logic made with AToM^3. Section 5 shows the transformation of the meta-model into a string grammar for the generation of the textual parser. Section 6 presents some related research, and finally section 7 ends with the conclusions and the future work.

2 Graph Grammars: An Informal Introduction

Graph grammars are a generalization of Chomsky grammar for graphs [15], devised in the seventies at TU Berlin by H. Ehrig and others. Graph grammars are composed of production rules, each having graphs in its left and right hand sides (LHS and RHS). In

the *Double Pushout Approach* (DPO), productions have the form: $p : L \xleftarrow{l} K \xrightarrow{r} R$, where *L* (left hand side), *K* (interface graph) and *R* (right hand side) are graphs and *l* and *r* are (usually injective) morphisms. That is, K is the set of nodes and edges which are preserved by the production, $L - K$ is the set of nodes and edges which are deleted and $R - K$ is the set of nodes and edges that are created by the production. The diagram in Figure 2 sketches the application of a rule on a graph G, resulting in graph H.

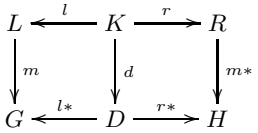

Fig. 2. Application of a Graph Gramamr Rule on a graph G.

Thus, in order to apply a production on a graph G, a match *m* should be found between the production's LHS L and the graph G. This can be either an injective or non-injective morphism. The next step is to delete all the elements in G matched with elements of $L - K$. Finally, the elements of $R - K$ are added. Note how this process can be expressed in terms of category theory as a double pushout (DPO) in category **Graph** [15]. Additionally, the double pushout approach needs two additional conditions, namely the *dangling* and the *identification* conditions. The *dangling condition* specifies that if an edge is not deleted its source and target nodes should be preserved. The *identification condition* specifies that if two nodes or edges in the interface are matched into a single node or edge in the host graph (via a non-injective morphism), then both should be preserved. In the single pushout approach (SPO) [15], a production is represented as a single mapping from L to R. If applied, dangling edges are deleted, and nodes or edges in conflict due to the identification condition are also deleted. Thus, in the SPO approach rules may have *secondary effects*.

Productions can be extended with *negative application conditions* (NAC), in such a way that for a production to be applied, there must not be a morphism between any of the production's NAC and the host graph. Thus, rules with NAC have the form: $p : N \xleftarrow{o} L \xleftarrow{l} K \xrightarrow{r} R$ (in the DPO approach).

3 AToM^3: A Brief Overview

AToM^3 [6] is a *multi-paradigm* tool (written in Python) that one of the authors developed in collaboration with McGill University in Montreal. The tool allows defining the abstract and concrete syntax of visual languages by means of meta-modelling. UML class diagrams and Entity Relationship (ER) meta-formalisms are available for meta-modelling, other formalisms and meta-formalisms can be described using them. Thus, from the (graphical) description of a meta-model (expressed in ER or UML classes) AToM^3 is able to generate a modelling tool for the described formalism. The meta-model describes the abstract syntax of the formalism. The concrete syntax is given by

assigning graphical attributes to the *entities* (or *classes*) and *relationships* (*or associations*) in the meta-model. Entities (or classes) can be provided with icon-like graphics, while relationships (*or associations*) can be assigned arrow-like graphics. An icon editor is available to design both kinds of graphics.

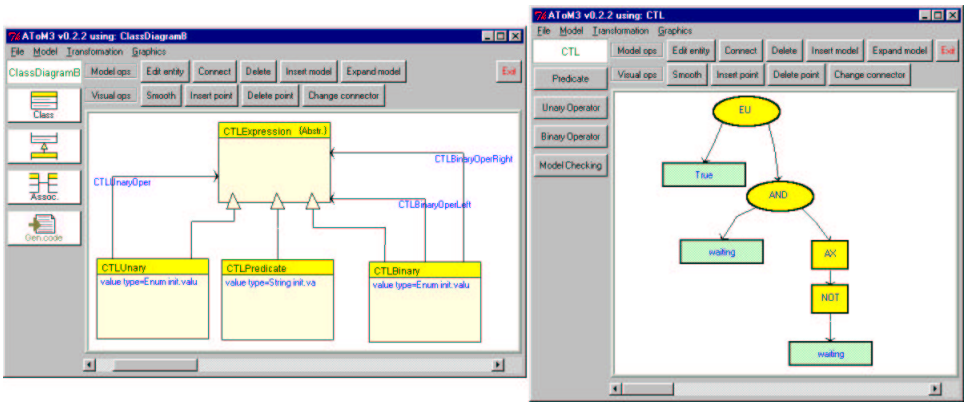

Fig. 3. CTL Meta-Model (left) and Generated Tool (right).

Figure 3 shows, on its left a meta-model for Computational Tree Logic [4] (CTL) defined with AToM^3, and on the right an automatically generated tool from the information in the meta-model. This example is discussed in more detail in next section. Note however that in this example the (graphical) concrete syntax is very simple, as *predicates* are assigned rectangles with the predicate name inside, *binary operators* are assigned ovals, with its type inside, and finally *unary operators* are assigned squares. More complex graphical concrete syntaxes can be defined with AToM^3, taking into account more advanced spatial relationships such as hierarchy or insideness (see for example [7]). Note also that the columns of buttons to the left of both windows are different. These buttons allow creating the elements of the formalism and are automatically generated from the meta-model of the formalism. This part of the user interface is indeed a model (under the *"buttons"* formalism), that the user can change. For example, it is possible to add extra buttons to perform computations on the current model.

In AToM^3, computations on models can be expressed either directly in Python, or using graph transformations. These are graphically modelled, and the AToM^3 graph grammar engine can be configured to work either in the *Single Pushout* or *Double Pushout* approaches [15]. To control the graph grammar execution, rules are assigned priorities (partial order), in such a way that the rules with the highest priority are checked first, if none of them makes a match, the control is passed to the set of rules in the next priority order. When a rule is executed, the control goes back to the set of rules with the highest priority. The execution ends when no rule could be applied in the set of lowest priority rules.

Additionally, the AToM^3 graph rewriting engine can be configured to allow (in the matching process) either an exact type matching between the nodes of the LHS and

the nodes in the host graph or a subtype matching. In the latter case AToM^3 checks at run-time whether the node (or the connection) in the host graph has at least the same set of attributes as the node in the LHS, that is, if the node in the host graph is a subtype of the node in the LHS. We do not need to express the subtyping relationship in the meta-models (in fact, if the Entity Relationship meta-formalism is used, we cannot include inheritance relationships in the meta-models), but this relationship is found at run-time. Finally, productions can be applied *step-by-step*, *animated* or in *continuous* mode. In the first mode, the rewriting engine waits for the user to press a button after each rule application. The second mode makes use of the *delay* attribute of rules to produce an animation of the graph grammar execution. This attribute can be changed by the rules. Finally, in the third mode, only the initial and the final graphs are shown.

4 An Example: Defining the Visual Computational Tree Logic Formalism

Computational Tree Logic (CTL) is a logic that allows expressing properties of computation paths. In particular it is widely used for model checking techniques [4]. Using CTL one can specify properties about the behaviour of a system. These properties are verified by a model checker against some representation of the system (a *Kripke structure*). We have implemented a simple explicit model checking algorithm in AToM^3 [5] that we use to check properties of reachability graphs obtained from a Petri net model. In its turn, this Petri net model could be the result of the transformation of several components of a *multi-formalism* system into a common semantic domain for analysis.

Thus, as we needed a tool inside AToM^3 for specifying CTL formulae, we meta-modelled CTL. This meta-model is shown to the left of Figure 3. A CTL expression (*CTLExpression* abstract class) is either a simple predicate (*CTLPredicate* class), an expression starting with a unary operator (*CTLUnary* class) or an expression starting by a binary operator (*CTLBinary* class). The meta-model also specifies the classes' attributes. Predicates have a name (attribute *value*, of type string), while unary and binary relationships have an attribute (called *value* in both cases) of enumerate type. This attribute stores the kind of operator: *AND*, *AU* (*for all paths ... until ...*) and *EU* (*exists one path such that ... until ...*) in the case of binary operators and *NOT*, *AX* (*in all paths, in next step...*) and *EX* (*exists a path, such that in next step...*) in the case of unary operators. This is a minimal set of operators, other ones can be defined using these. We have defined three special predicates: *True*, *False* and *Deadlock*. The latter evaluates to true in nodes without outgoing transitions.

Right of Figure 3 shows a valid instance of the previous meta-model, which expresses the property *E(True U (waiting AND AX NOT waiting)*. The tool in which the model is built is AToM^3 itself, loaded with the CTL meta-model defined on the left of the figure. This tool is used by the model checker built in AToM^3 to specify CTL formulas. This meta-modelling solution is effective in the sense that one can generate a visual modelling tool in very little time. But unlike for other graphical formalisms, CTL formulae can be more naturally specified in textual form. Thus, one would like a tool to process textual formulas in CTL. Note how, one could think of the meta-model as describing the structure of the *abstract syntax graphs* produced by a parser of the

textual notation. Our aim is to automate as much as possible the generation of such a parser given the meta-model.

This is just a particular example of a more general problem that we face in multi-formalism modelling, where we specify a composite system using several formalisms. Some of them may be graphical, while others may be textual. Thus, one would like a general means to specify and process both kind of formalisms. Our solution relies on meta-modelling to describe the *abstract syntax* of the (graphical and textual) formalism. In an ideal solution one could define mappings to graphical or textual concrete syntaxes. In the more general case, one could even define different (graphical or textual) concrete syntax mappings for a given formalism.

5 Transforming Simple Meta-Models into String Grammars

For the purpose of processing textual languages, our approach is to define them with a meta-model, and then transform them into the *"string grammars"* formalism, also defined with a meta-model. The string grammars we consider are composed of *productions* with a unique non-terminal symbol in the left hand side (LHS) and several terminal or non-terminal symbols in the right hand side (RHS). This meta-model is depicted to the left of Figure 4, enclosed in the rectangle labelled as *"String Grammars Meta-Model"*. Abstract class *Symbol* has an attribute called *value* of type string. In the case of a terminal symbol (*TSymbol*), it contains a regular expression. On the contrary, if the symbol is non-terminal (*NTSymbol*), it contains the name of the symbol. Note how (symbols in the *production*'s RHS) can be connected via the *RHSnext* relationship.

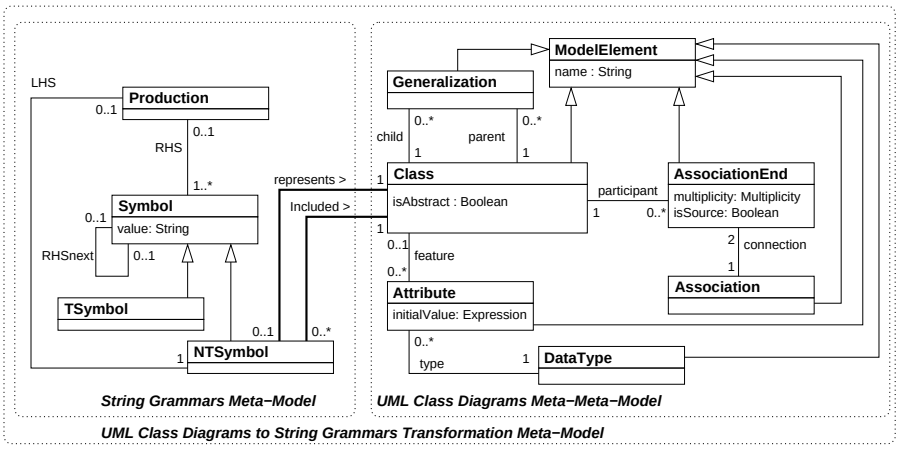

Fig. 4. Meta-Model for the Transformation from UML Simple Meta-Models into String Grammars.

The model to the right of Figure 4 shows a subset of the UML meta-metamodel (see [18]) that we use in AToM^3 for the modelling of formalisms. It simply defines

classes composed of a number *Attributes* (their possible types are not shown for simplicity). *Classes* can be connected through *Generalization*, or via simple *Associations*. These have two *Association Ends*, attributed with the multiplicities. The window to the left of Figure 3 shows an instance of this meta-metamodel, although once the meta-metamodel has been provided with a *concrete syntax*.

As stated before, we use graph grammars to describe the transformation of models between formalisms. During the transformation process, the model becomes a blend of elements of the source and target formalisms, but at the end of the transformation, the model should be entirely described in the target formalism. Thus, we also need a meta-model describing the kind of models that can arise during the transformation [8]. In our example, for transforming models defined with this subset of UML into string grammars, we need to define a meta-model combining both formalisms. This meta-model is the disjoint union of both formalisms, plus some extra auxiliary elements needed for the transformation. These are typically relationships relating elements of the *source* (UML meta-models) and *target* (string grammars) formalisms. In the figure, these extra elements are relationships *"represents"* and *"included"*. Relationship *"represents"* models the fact that we assign a production (its LHS) to each *class*. Relationship *"included"* models the fact that a *class* can be included in the RHS of a production.

Figure 5 shows some of the graph grammar rules for the transformation of simple UML meta-models into string grammars. In particular, the models we can transform by now are formed by a single abstract class (the initial symbol), with a number of child classes. These are considered tokens if they are unconnected, unary operators if they have a single outgoing association, or binary operators if they have two outgoing associations. Note how, although very simple, many textual formalisms can be described in this way (for example, algebraic equations and several kinds of logic). In particular, the meta-model for CTL presented in Figure 3 falls into this category of models.

Rule 1 creates a *production* for each abstract class found in the meta-model. The negative application condition (NAC) in the rule prohibits the application of the rule if the class has been already assigned a *production*. In all the rules, we have indicated the type of each node besides it in italics, and used the abbreviations *"Rep"* and *"Incl"* for relationships *"represents"* and *"included"* respectively. We use numbers to represent the morphisms between NAC, LHS and RHS. That is, nodes and edges with the same number in LHS and RHS belong to the interface graph K (see Figure 2) and represent the same element in the host graph.

Rule 2 creates right hand sides of *productions* for each child class of an abstract class with no associations. As stated before, these classes represent the tokens, which will be defined by another graph grammar rule (not shown in the picture). Rule 3 is similar to the previous one but creates the right hand side of the string production for *unary operators* (classes with exactly one association). Note how in this case, the associated class is added as the second symbol of the production right hand side (through the *RHSnext* relationship). Finally, additional rules (not shown in the figure) create *productions* for binary operators, and for the description of tokens and operators. Classes representing unary and binary operators are supposed to have a distinguished attribute called *value*, of type enumerate, which contains the allowed operators. In our example, the unary operators are *NOT*, *AX* and *EX* and the binary operators *AND*, *AU* and *EU*.

For tokens (in our case class *Predicate*), the *value* attribute can be *string*, *integer* or *float*, in such a way that a regular expression accepting these data types is generated for them. Furthermore, in the case of a *string* type, we set the regular expression to accept only valid identifiers (those starting by a letter). Finally, some other rules remove the elements of the UML formalism, in such a way that at the end of the transformation, only elements of the *string grammar* formalism remain.

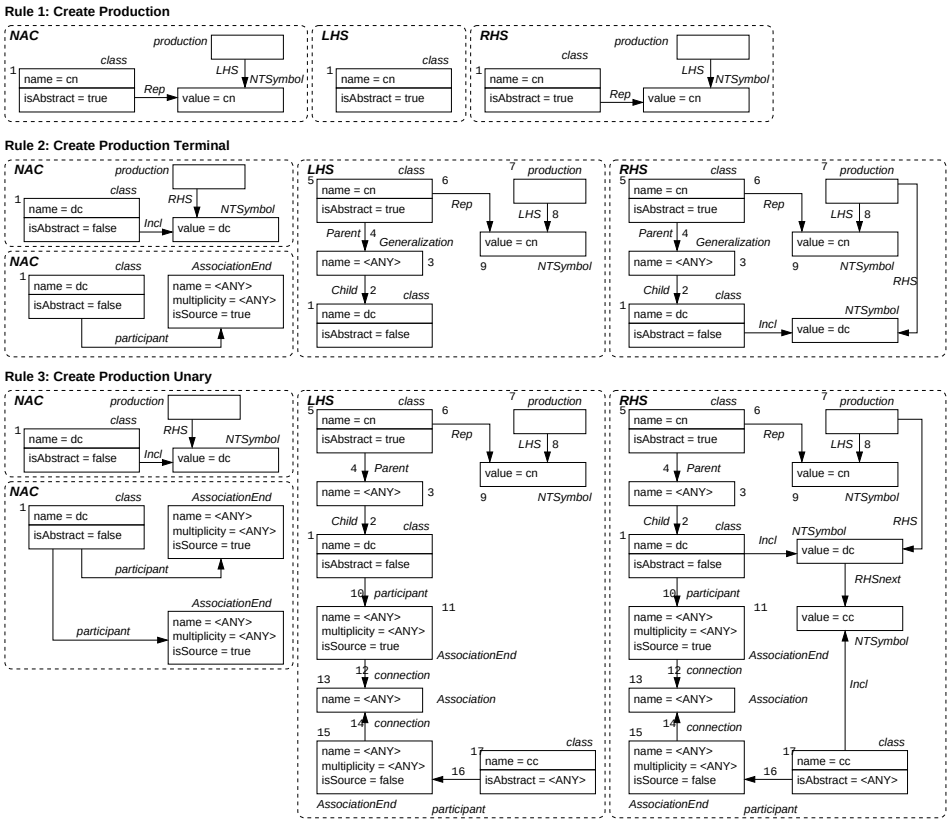

Fig. 5. Some Rules for the Transformation of Simple Meta-Models into String Grammars.

Figure 6 shows the resulting model (in the *string grammars* formalism) once we apply the previous transformation rules to the CTL meta-model shown in Figure 3. Note how the resulting grammar is left-recursive. This can be eliminated implementing the algorithm for eliminating left recursion as a graph grammar. Once the transformation has been performed, it is easy to dump the *string grammar* model into a file (by traversing all the *productions*, an excerpt is shown in Figure 7), to be processed by some parser-generating program. In our case, we used TRAP [9] to generate a parser for the CTL grammar. The reason for choosing TRAP is that it is able to generate Python parsers, easy to integrate with AToM^3, also written in Python. The excerpt shown in Figure 7

shows three of the string grammar rules. The first one (*CTLPredicate*) just describes the *String* type by means of a regular expression (as the *value* attribute of *CTLPredicates* is of type *String*). The second one (*CTLUnary*) has three right hand sides, and describes the three unary operators. The AToM^3 model is built by means of semantic actions that invoke the AToM^3 API for creating objects (*createNewCTLUnary*). The actions associated with a string grammar are executed whenever the production is used, and build the *abstract syntax graph* according to the meta-model. The last production (*CTLBinary*) is similar to the previous one, but describes the three binary operators.

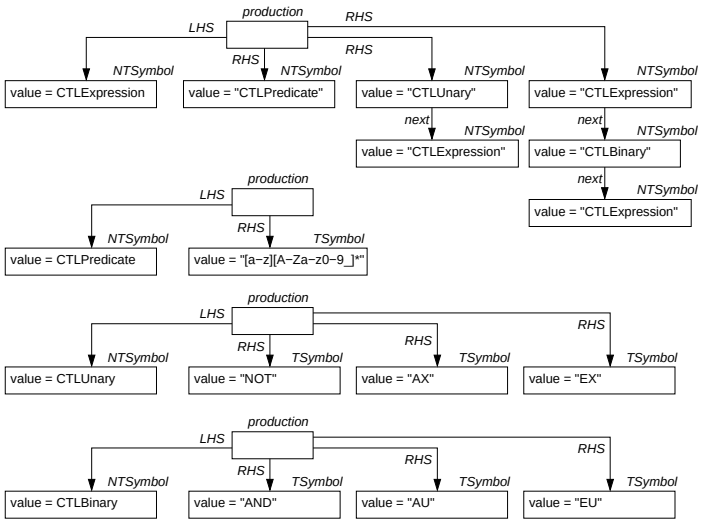

Fig. 6. String Grammar Model Resulting from the Transformation of the CTL Meta-Model.

In AToM^3, graph grammar rules can have actions (specified in Python) that are executed if the rule is applied. We have provided the graph grammar rules with actions to annotate the *productions* and *NTSymbols* of the string grammar model with semantic actions that call the AToM^3 API (*createNewCTLUnary* and *createNewCTLBinary* in Figure 7.)

The textual parser was integrated into AToM^3 in order to perform (explicit) model checking of systems described as Kripke structures. In particular we have used the model checker to verify properties of physical [5] (described with Statecharts and Petri-Nets) as well as logical systems (described using different UML diagrams).

6 Related Work

Other approaches – such as for example in Glide [10] – go the other way around: the definition of the languages is done through a textual grammar (in the BNF style), and optionally, a concrete visual syntax can be assigned to the language. This approach has

```
compiler CTL

token CTLPredicate '[a-z][A-Za-z0-9_]*'

# This production is called CTLUnary and returns an object of type CTLUnary
nterm CTLUnary::CTLUnary
  <- "NOT" :
       # Semantic action to create (and return) a CTLUnary object of type 'NOT'
       at3.createNewCTLUnary('NOT')

  <- "AX" :
       # Semantic action to create (and return) a CTLUnary object of type 'AX'
       at3.createNewCTLUnary('AX')

  <- "EX" :
       # Semantic action to create (and return) a CTLUnary object of type 'EX'
       at3.createNewCTLUnary('EX')

# This production is called CTLBinary and returns an object of type CTLBinary
nterm CTLBinary::CTLBinary
  <- "AND" :
       # Semantic action to create (and return) a CTLBinary object of type 'AND'
       at3.createNewCTLBinary('AND')

  <- "AU" :
       # Semantic action to create (and return) a CTLBinary object of type 'AU'
       at3.createNewCTLBinary('AU')

  <- "EU" :
       # Semantic action to create (and return) a CTLBinary object of type 'EU'
       at3.createNewCTLBinary('EU')
...
```

Fig. 7. Excerpt of the TRAP Grammar Created from the Model in Figure 6.

the disadvantage that it is not very appropriate for purely visual languages, which are more naturally described through a graphical meta-model.

The approach taken in the ESPRESS project [3] is to integrate heterogeneous modelling, analysis and simulation tools by defining *adaptors* for them, that should be coded by hand. For the semantic integration of the tools, the different formalisms are translated into a common notation. In this paper we deal with the automatic generation of textual parsers given a meta-model of the formalism. The idea is that once the programs are successfully parsed, they are converted into *abstract syntax graphs*, valid instances of the original meta-model. That is, we can think of this work as a means to automatically obtain basic *adaptors* for tools based on textual formalisms. Additionally, in our work translators between formalisms are specified using graph grammars.

Other possibility is to use an idea similar to pair grammars [14], where a graph grammar rule is assigned to each string grammar production. Whenever a string production is used, the associated graph grammar rule is executed. This is a way to specify a string to graph translation and allows building an instance model of some meta-model starting from a string grammar. The drawback is that one has to code by hand the string grammar, the graph grammar for the generation of the visual language and the meta-model. Our aim is to automate the whole process in such a way that one only has to provide the meta-model. We could have still applied this approach by assigning graph grammar rules to the automatically generated productions in Figure 7 (instead of calling the AToM^3 API to build the model), but in this case, we would have to create the generation graph grammar for the visual CTL.

With respect to the specification of transformations, other approach is the use of triple graph grammars [16]. Using triple graph grammars one can obtain translators from the source to the target formalism and vice versa with the same triple graph

grammar. The approach is mostly useful for syntax-directed environments, in which the editing actions are specified by means of graph grammar rules. In AToM^3 this is not the default approach, as the user can create and connect entities of the formalism, and model correctness is guaranteed by constraints (defined in the meta-model) that are evaluated when certain events are triggered. Additionally, in AToM^3 one could define graph grammar rules to model the editing actions, although as stated before, this is not necessary. With triple graph grammars, while the user is building a model in the source formalism, an equivalent model is created at the same time in the target formalism. It is not straightforward to use this approach to translate an existing model, as in this case the graph grammar rules must be *monotonic* (that is any production's left-hand side must be part of its right-hand side [16]). Additionally AToM^3 does not have the capability to work with triple graph grammars.

For a formalism transformation to be really useful, one has to proof several properties, such as confluence (that the result of the transformation is deterministic), termination, consistency (the resulting models are valid instances of the target formalism meta-model) and behavioural equivalence of source and target models. Some initial work in the verification of these properties has already been done [8], although it is out of the scope of this paper.

To the knowledge of the authors, other well-known tools [1][12] in the graph grammars community for the definition of visual languages do not consider by now the possibility to integrate textual languages.

7 Conclusions

In this paper we have presented our approach for integrating visual and textual languages in a uniform framework. The approach consists on defining both kinds of languages by means of meta-modelling. In the case of a visual language, a graphical concrete syntax is assigned. In the case of a textual language, the meta-model is translated into the "*string grammars*" formalism, and a parser is then obtained that transform textual programs into instances (*abstract syntax graphs*) of the textual language meta-model. Once textual and graphical models are expressed as attributed, typed graphs, they can be manipulated using graph grammars. We have presented an example in which we have created textual and graphical parsers for Computational Tree Logic for their integration in a model checking engine for AToM^3.

The work presented in the paper is in its initial state, although we have successfully used these ideas to generate the visual and textual parser for CTL. At present, we are finishing the actual implementation of the *string grammars* meta-model and the rules for its manipulation. We want to use these ideas to generate visual and textual parsers for other formalisms, such as differential algebraic equations.

There are additional problems to be solved, for example the issue of operator's precedence. This precedence is not explicitly specified if the model is built graphically, but it is realized by traversing the model. On the contrary, it may have to be specified with parenthesis if we specify the model in a textual way. Thus, parenthesis may have to be automatically added in the *string grammar* model.

Finally, we are also working in integrating parallel graph grammars [17] into AToM^3. This is an extension of graph grammars for coordinating the parallel application of different subproductions.

Acknowlegdments

We would like to aknowledge the Spanish Ministry of Science and Technology (project TIC2002-01948) for partially supporting this work.

References

1. Bardohl, R., Ermel, C., Weinhold, I. 2002 *AGG and GenGED: Graph Transformation-Based Specification and Analysis Techniques for Visual Languages* In Proc. GraBaTs 2002, Electronic Notes in Theoretical Computer Science 72(2).
2. Brooks, F. P. 1995. *The Mythical Man Month*. Addison-Wesley.
3. Büssow, R., Grieskamp, W., Heicking, W. 1998. *An Open Environment for the Integration of Heterogeneous Modelling Techniques and Tools* International Workshop on Current Trends in Applied Formal Methods; Boppard Germany LNCS 1641. Springer. See also the ESPRESS project home page: `http://www.first.gmd.de/~espress`
4. Clarke, E. M., Grumberg, O., Peled, D. A. 1999. *Model Checking*. MIT Press.
5. de Lara, J., Guerra, E., Vangheluwe, H. 2003. *Meta-Modelling, Graph Transformation, and Model Checking for the Analysis of Hybrid Systems*. To appear in Proceedings of AGTIVE'2003 (Applications of Graph Transformation with Industrial Relevance). Charlottesville (USA).
6. de Lara, J., Vangheluwe, H. 2002 *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* In European Conferences on Theory And Practice of Software Engineering ETAPS02, Fundamental Approaches to Software Engineering (FASE). LNCS 2306, pp.: 174 - 188. Springer-Verlag. See also the AToM^3 home page: `http://atom3.cs.mcgill.ca`.
7. de Lara, J., Vangheluwe, H. 2002 *Computer Aided Multi-Paradigm Modelling to process Petri Nets and Statecharts.* ICGT'2002. LNCS 2505. Pp.: 239-253.
8. de Lara, J., Taentzer, G. 2003. *Automated Model Transformation and its Validation with AToM^3 and AGG*. Submitted to Diagrams'04.
9. Ernst, T. 1999. *TRAPping Modelica with Python* In Proc. ETAPS/CC'99. LNCS 1575, pp.: 288-291. Springer-Verlag. See also: `http://www.first.gmd.de/smile/trap`.
10. Kleyn, M. F. E. 1995. *A High Level Language for Specifying Graph-Based Languages and their Programming Environemnts*. PhD. Thesis in the University of Texas at Austin.
11. Lédeczi, A., Bakay, A., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G. 2001. *Composing Domain-Specific Design Environments.* IEEE Computer, Nov. 2001, pp.: 44-51. See also the GME home page: `http://www.isis.vanderbilt.edu/Projects/gme/default.html`, Vanderbilt University.
12. Minas, M. 2002. *Specifying Graph-like diagrams with DIAGEN* Science of Computer Programming 44, pp.: 157-180.
13. Pohjonen, R., Tolvanen, J-P. 2002. *Automated Production of Family Members: Lessons Learned*, Proceedings of the Second International Workshop on Product Line Engineering - The Early Steps: Planning, Modeling, and Managing (PLEES'02), pp.: 49–57. See also the MetaEdit+ home page. `http://www.metacase.com`, MetaCase Consulting.

14. Pratt, T. W. 1971. *Pair Grammars, Graph Languages and String-to-Graph Translations.* Journal of Computer and System Sciences, Vol.5, pp.:560-595.
15. Rozenberg, G. (ed) 1997. *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol 1. World Scientific.
16. Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars*. In Ernst W. Mayr, Gunther Schmidt, Gottfried Tinhofer (Eds.): Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94. Lecture Notes in Computer Science 903, pp.: 151-163. Springer.
17. Taentzer, G. 1996. *Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems*. PhD. Dissertation, Shaker Verlag.
18. UML 1.5 specification by OMG: `http://www.omg.org/technology/documents/formal/uml.htm`.
19. Vangheluwe, H., de Lara, J., Mosterman, P. 2002. *An Introduction to Multi-Paradigm Modelling and Simulation.* In Proceedings of AI, Simulation and Planning - AIS'2002. Pp.: 9-20. Lisbon. SCS International.
20. Warmer, J. B., Kleppe, A. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Object Technology Services. See also the OCL specification at the OMG home page: `http://www.omg.org`