Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

COMPUTER AUTOMATED MULTI-PARADIGM MODELLING: META-MODELLING AND GRAPH TRANSFORMATION

Hans Vangheluwe

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

Juan de Lara

ETS Informática
Universidad Autonóma de Madrid
Madrid 28049, SPAIN

ABSTRACT

We present Computer Automated Multi-Paradigm Modelling (CAMPaM) (Mosterman and Vangheluwe 2002) for Model-Driven Development based on Meta-Modelling and Graph Transformation. The syntax of a class of models of interest is graphically meta-modelled in an appropriate formalism such as Entity-Relationship Diagrams. From this description of abstract syntax, augmented with concrete (visual) syntax information, an interactive, visual modelling environment is automatically generated. As the abstract syntax of models, irrespective of the formalism they are described in, is graph-like, graph rewriting can be used to perform model transformation. Graph Grammar models thus allow for model transformation specification. The Graph Grammar formalism can be meta-modelled in its own right and hence a visual environment for manipulating transformation models can also be automatically generated. Graph rewriting provides a rigourous basis for specifying and analyzing model transformations such as simplification, simulation, and code generation. In this article, we introduce AToM^3, A Tool for Multi-formalism and Meta-Modelling. We present the meta-modelling and graph transformation concepts through a simple reactive system example: a Timed Automata model of a traffic light. Meta-modelling Timed Automata, generating the visual modelling environment, and modelling transformations as graph grammers, as well as executing them, are all performed in the AToM^3 environment. The model transformations include simulation, transformation into Timed Transition Petri Nets, and code generation.

1 INTRODUCTION

Computer Automated Multi-Paradigm Modelling (CAMPaM) aims to simplify the modelling of complex systems by combining three different directions of research:

- *Meta-Modelling*, which is the process of modelling formalisms.
- *Model Abstraction*, concerned with the relationship between models at different levels of abstraction.
- *Multi-Formalism modelling*, concerned with the coupling of and transformation between models described in different formalisms.

In the sequel, we will focus on meta-modelling and on graph grammars to model transformations needed for model abstraction and multi-formalism modelling.

Meta-modelling can help in defining high abstraction level notations. With meta-modelling, we can describe, using a high-level, graphical notation, the (possibly graphical) syntax of languages for particular needs (i.e., Domain Specific Visual Languages). Such descriptions are called meta-models. Some languages – such as the UML – are rigourously defined through meta-modelling. But meta-modelling the syntax of a language is only one side of the coin. One needs to formally specify the semantics of a language. For example, we may be interested in defining a language's *operational semantics* (how models described in the language are going to be executed), its *denotational semantics* (defining a mapping onto another well-defined language; this may include code generation when mapping onto a virtual machine), or *optimizing* (reducing the complexity without removing salient features) the models. As models, meta-models and meta-metamodels can all be described as attributed, typed graphs, in this paper we present a formal, graphical and high-level notation to specify model manipulations: graph grammars (Ehrig et al. 1999). In the graph grammar formalism, computations on models can be explicitly modelled.

We implemented these concepts in a tool called AToM^3, A Tool for Multi-formalism and Meta-Modelling. AToM^3's design has been previously described in de Lara and Vangheluwe (2002a) and de Lara et al. (2002). We follow the maxim *everything is a model*. That is, not only formalisms and computations are modelled explicitly, but also composite types and the user interfaces of the generated tools. The tool was bootstrapped from a small kernel with

code-generating capabilities, and from models for various parts of AToM^3.

In the rest of the paper, we will clarify the meta-modelling and graph transformation concepts through an example: (meta-)modelling the *Timed Automata* formalism and defining different kinds of transformations, including simulation, transformation into Timed Transition Petri Nets (Peterson 1981) for subsequent analysis, and code generation for a particular application. We present a small traffic light example. The traffic light is initially in the flashing yellow mode. A policeman interrupt brings the system in the yellow (not flashing) mode. From there, the system cycles through the modes red, green, and yellow. Upon entering each of these modes, the traffic light displays the appropriate colour. The system stays in each of these modes for 5, 20, and 35 seconds respectively. From each of these modes, a policeman interrupt brings the system back in the flashing mode. In the green mode, a pedestrian interrupt will instantly bring the system in the yellow mode (thus shortening the time a pedestrian needs to wait). All other modes ignore the pedestrian interrupt.

2 META-MODELLING

When modelling complex physical or logical systems it is desirable to use the most appropriate formalism to optimally describe different aspects or components of the system. In this case, one has to solve the problem of building and interconnecting a plethora of different tools, each one especially built for each formalism. *Meta-Modelling* alleviates these problems. By means of meta-modelling one can describe, usually using graphical, high-level modelling notations –meta-formalisms– such as UML class diagrams or Entity-Relationship Diagrams, the family of models one is interested in processing. This description of the formalism's syntax is called a *meta-model*. Similarly, a model of a meta-formalism is called a *meta-meta-model*, and a model built using a formalism is simply called a *model*.

To be able to fully specify modelling formalisms, the meta-formalism may have to be extended with the ability to express constraints. For example, when modelling *Deterministic Finite state Automata*, different transitions leaving a given state must have distinct labels. This cannot be expressed using Entity-Relationship Diagrams alone. Expressing constraints becomes possible by adding a constraint language (usually with a textual syntax) to the meta-formalism (usually with a graphical syntax). Some systems (Sztipanovits et al. 1995) use the Object Constraint Language OCL used in the UML. As AToM^3 is implemented in the scripting language Python, arbitrary Python code may also be used.

The advantage of meta-modelling is that it dramatically enhances the productivity in creating custom modelling environments (domain-specific tools). On one hand, once the meta-model is defined, meta-modelling tools are able to automatically produce a modelling tool for the defined formalism. On the other hand, once the meta-model is defined, it is easy to make small modifications to obtain customized variations of the modelling formalism for specific user groups.

As an example, we demonstrate how to build a meta-model for the Timed Automata formalism with AToM^3. In AToM^3, the default meta-formalism is Entity-Relationship Diagrams. To define the meta-model, one has to provide abstract syntax (denoting entities of the formalism, their attributes, relationships and constraints) as well as concrete graphical syntax (how the entities and relationships should be rendered in a visual interactive tool, and possible graphical constraints) information. Once the formalism is modelled, AToM^3 generates Python code that can be loaded by the AToM^3 kernel. Then the tool accepts valid models according to the compiled formalism definition. Using AToM^3, the effort to produce a customized visual modelling tool may be reduced to just a few hours for typical formalisms.

The meta-model for Timed Automata is shown in the upper-left window in Figure 1.

Note how a Timed Automata model is composed of states (which have a name) and transitions between them. We have defined two kinds of transitions. We use the first kind (named *TATimedTransition*) to specify a delay as the only condition for state change. We use the second kind (named *TATransition*) to specify that a certain input must be present for a state change to occur. This input must occur *before* the minimum delay that triggers a state change has occurred. We have included an entity called *TACurrent* which points to the initial state by means of the relationship *TAPointsTo*. During the simulation of a model, this points to the current state. This entity also has attributes which store the current simulation time and the last time a transition occurred. For simulation purposes, we include a sequence of scheduled (input) events in the model. We represent the input as a linked list of *TAinput* entities, whose first element is linked to the *TACurrent* entity. Inputs are provided with a value and a time stamp reflecting the time at which they are scheduled to occur. The input with the special value *End* denotes the end of the list. A time stamp -1.0 in *End* denotes plus infinity. Any non-negative time in *End* specifies when to halt the simulation.

We also define the graphical appearance of these entities and relationships, declare global attributes (such as the model name, author, input alphabet, etc.) as well as constraints. We must for example specify that exactly one *TACurrent* entity is allowed in a model and that each state has at most one outgoing timed transition.

AToM^3 compiles the meta-model. After loading this compiled meta-model, AToM^3 only accepts syntactically correct models in the Timed Automata formalism. The right window in Figure 1 shows a model built using this

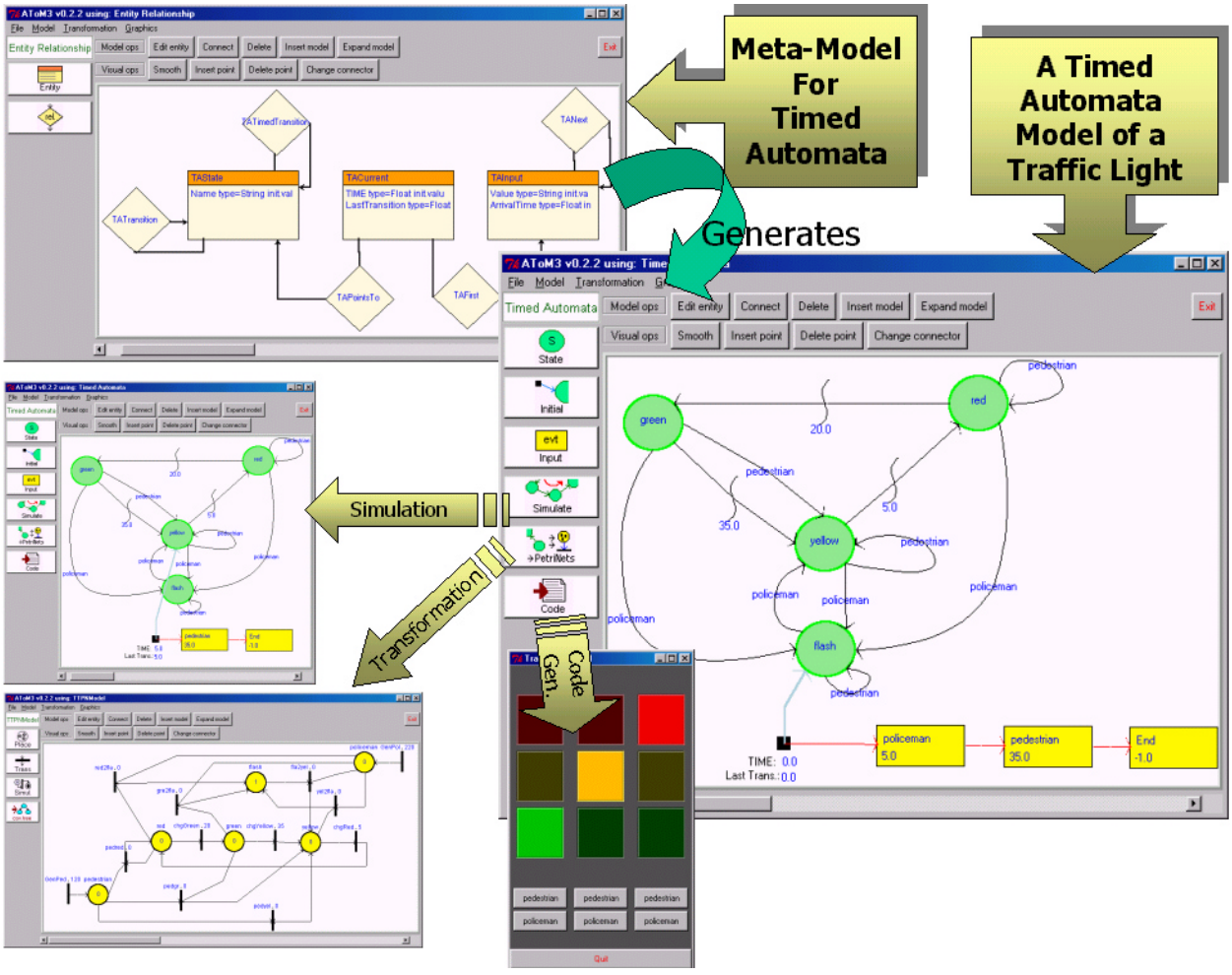


Figure 1: Meta-Modelling Timed Automata with AToM^3

formalism. Note how the column of buttons on the left has changed compared to ones that appear when the Entity-Relationship Diagrams formalism is loaded. The buttons available in the user interface are described in the *Buttons* formalism. This model is generated by AToM^3 based on the entities and relationships in a formalism's meta-model. The user may customize this model. In this example, we have modified the model, deleting buttons to create transitions (as we can create them implicitly by connecting states) and added some other buttons to execute graph grammar models on the current model (to simulate, transform into Petri Nets, and generate code). This is explained in the next section.

3 GRAPH TRANSFORMATION

The *transformation* of models is a crucial element in model-based endeavours. As models, meta-models and meta-meta-models are all in essence attributed, typed graphs, we can transform them by means of graph rewriting. The rewriting is specified in the form of graph grammar (Ehrig et al. 1999) models. These are a generalization, for graphs, of Chomsky grammars. They are composed of rules. Each rule consists of left hand side (LHS) and right hand side (RHS) graphs. Rules are evaluated against an input graph, called the host graph. If a matching is found between the LHS of a rule and a sub-graph of the host graph, then the rule can be applied. When a rule is applied, the matching subgraph of the host graph is replaced by the RHS of the rule. Rules can have applicability conditions, as well as actions to be performed when the rule is applied. Some graph rewriting systems have control mechanisms to determine the order in which rules are checked. In AToM^3, rules are ordered according to a user-assigned priority, and are checked from higher to lower priority. After a rule matching and subsequent application, the graph rewriting system starts the search again. The graph grammar execution ends when no more matching rules are found.

We are interested in three kinds of transformations of our example model. The first is model execution (defining the operational semantics of the formalism). The second is model transformation into another formalism (expressing the semantics of models in one formalism by mapping onto a

known formalism). A special case of this is when the target formalism is textual. In this case it is possible to describe by means of meta-modelling, the *Abstract Syntax Graph* of the textual formalism (that is, the intermediary representation used be compilers once they parse a program in text form), in such a way that models in textual formalisms can then be processed as graphs. The third one is model optimization, for example reducing its complexity (maintaining pertinent invariants however).

On one hand, graph grammars have some advantages over specifying the computation to be done in the graph using a traditional programming language. Graph grammars are a natural, formal, visual, declarative and high-level representation of the computation. Computations are thus specified by means of high-level models, expressed in the graph grammar formalism. The theoretical foundations of graph rewriting systems may assist in proving correctness and convergence properties of the transformation tool. On the other hand, the use of graph grammars is constrained by efficiency. In the most general case, subgraph isomorphism testing is NP-complete. However, the use of small subgraphs on the LHS of graph grammar rules, as well as using node and edge types and attributes can greatly reduce the search space. This is the case with the vast majority of formalisms we are interested in. It is noted that a possible performance penalty is a small price to pay for explicit, reusable, easy to maintain models of transformation. In cases where performance is a real bottleneck, graph grammars can still be used as an executable specification to be used as the starting point for a manual implementation.

3.1 Simulating Timed Automata

As an example, Figure 2 shows a graph grammar which models a simulator for the Timed Automata formalism defined before. The grammar is composed of four rules. The first changes the current state due to a timed transition if the actual time plus the transition delay is less than the time at which the first event in the scheduled event list occurs. Note how nodes and connections in LHSs and RHSs are identified by means of labels (numbers). If a number appears on both the LHS and the RHS of a rule, the node or connection is retained when the rule is applied. If the number appears only in the LHS, the node or connection is deleted when the rule is applied. Finally, if the number appears only in the RHS, the node or connection is created when the rule is applied. Node and connection attributes in LHSs must be provided with attribute values which will be compared with the node and connection attributes of the host graph during the matching process. These attributes can be set to `<ANY>` or have specific values.

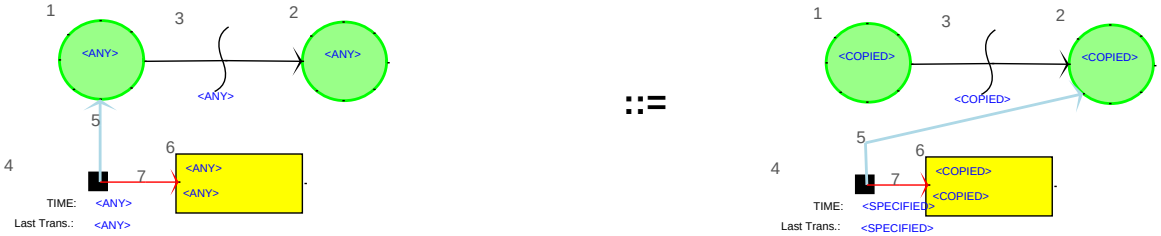
In the first rule, we have set all the attributes of the nodes and connections to `<ANY>`. In the RHS, we can specify changed attribute values for the nodes that also appear in the LHS. Obviously, we must specify the attribute values of the newly created nodes or connections. In AToM^3, we can either copy the value of the attributes of the LHS (this appears as `COPIED` in the figure), specify a new value, or associate arbitrary Python code to compute the attribute value, possibly based on other nodes' attributes. The second rule is similar to the first one, but handles the case when the timed transition departs from and arrives at the same state. The third rule deals with the case of transitions due to input (and not due to a time delay). In this case, the input is consumed (note how node 6 and connection 9 do not appear in the RHS) and the current state is changed. The last rule is similar to the previous one, but deals with the case of a transition which departs from and arrives at the same state.

A snapshot of the simulation is shown in the small window of AToM^3 pointed to by the arrow *Simulation* in Figure 1. In Figure 3, a trace of a typical simulation is shown in detail. The initial time is `0.0` and the automaton is in the `flash` state. The time-ordered list on the bottom right shows the scheduled inputs. At time `5.0`, the `policeman` interrupt event brings the automaton in the `yellow` state. From now on, the system is in autonomous mode and makes a transition to the `red` state `after 5.0`, i.e., at time `10.0`. After another `20.0`, at time `30.0`, the automaton is brought in state `green`. The next autonomous transition would occur `35.00` later, but is pre-empted by the external `pedestrian` interrupt at time `35.0`, bringing the automaton in state `yellow` at time `35.0`. From there, the autonomous behaviour (the `yellow, red, green` cycle) resumes.

3.2 Transformation for Analysis

In the real-time application which will ultimately be generated from our model, it is possible for the user to generate, at *any* time, any of the model's possible input events. This is in contrast with the single specific list of scheduled input events used for simulation. A symbolic analysis of model properties such as reachability of certain states may thus be useful. To get insight into those properties of our model, we transform it into a Timed Transition Petri Nets. The transformation is again specified using a graph grammar model. It converts states into places, with a token in the initial state, input events into places, and transitions into Petri Net transitions as described in Murata (1989). The transformation is a variant of the more general transformation of Statecharts into Petri Nets described in de Lara and Vangheluwe (2002b). Once the model is transformed into a Timed Transition Petri Net, we can use the available analysis and simulation techniques (Peterson 1981), such as the ones based on the matrix of equations, structural properties, or the reachability graph. This last technique allows us to investigate whether the system may deadlock, reaches a

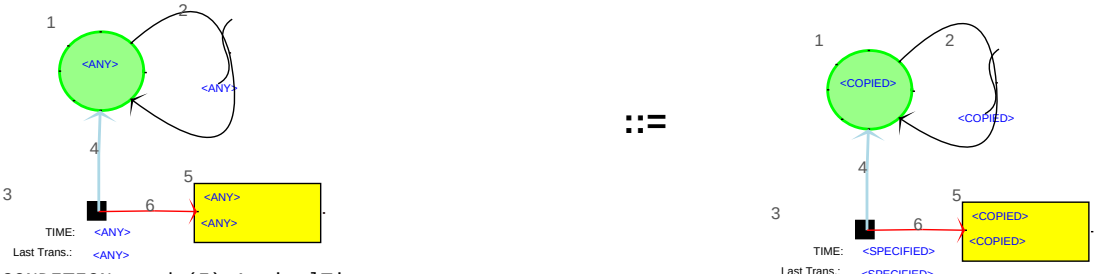
TA_MoveTimeTrans **Priority 1**




```
CONDITION: node(6).ArrivalTime >
           node(4).Time + node(3).Time
```

```
TIME += node(3).Time
Last Trans = TIME
```

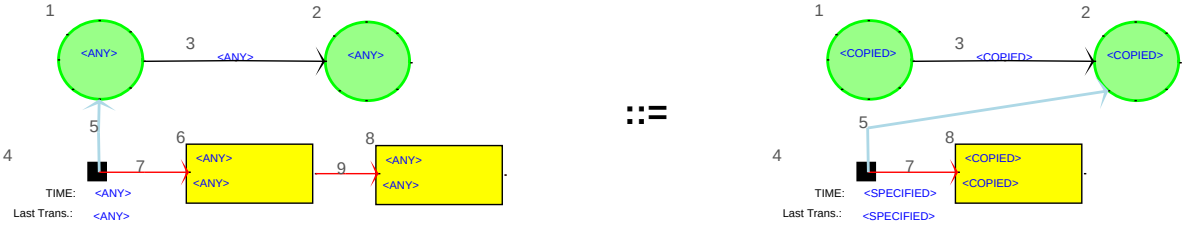
TA_MoveTimeTransSelf **Priority 2**




```
CONDITION: node(5).ArrivalTime >
           node(3).Time + node(2).Time
```

```
TIME += node(2).Time
Last Trans = TIME
```

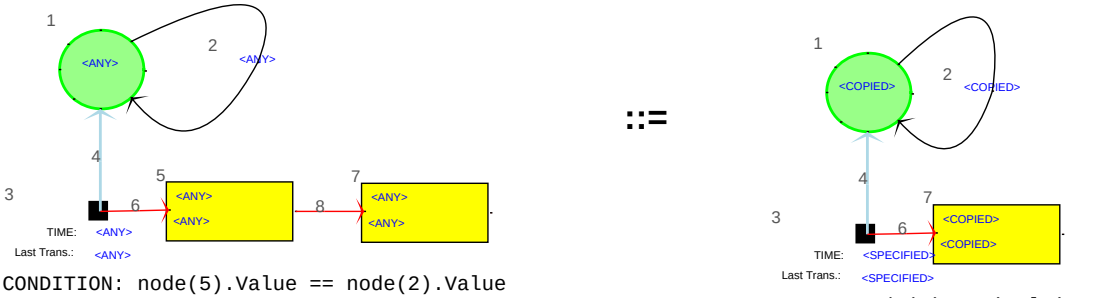
TA_MoveTrans **Priority 3**




```
CONDITION: node(6).Value == node(3).Value
```

```
TIME = node(6).ArrivalTime
Last Trans = TIME
```

TA_MoveTransSelf **Priority 4**




```
CONDITION: node(5).Value == node(2).Value
```

```
TIME = node(5).ArrivalTime
Last Trans = TIME
```

Figure 2: A Graph Grammar Model of a Timed Automata Simulator

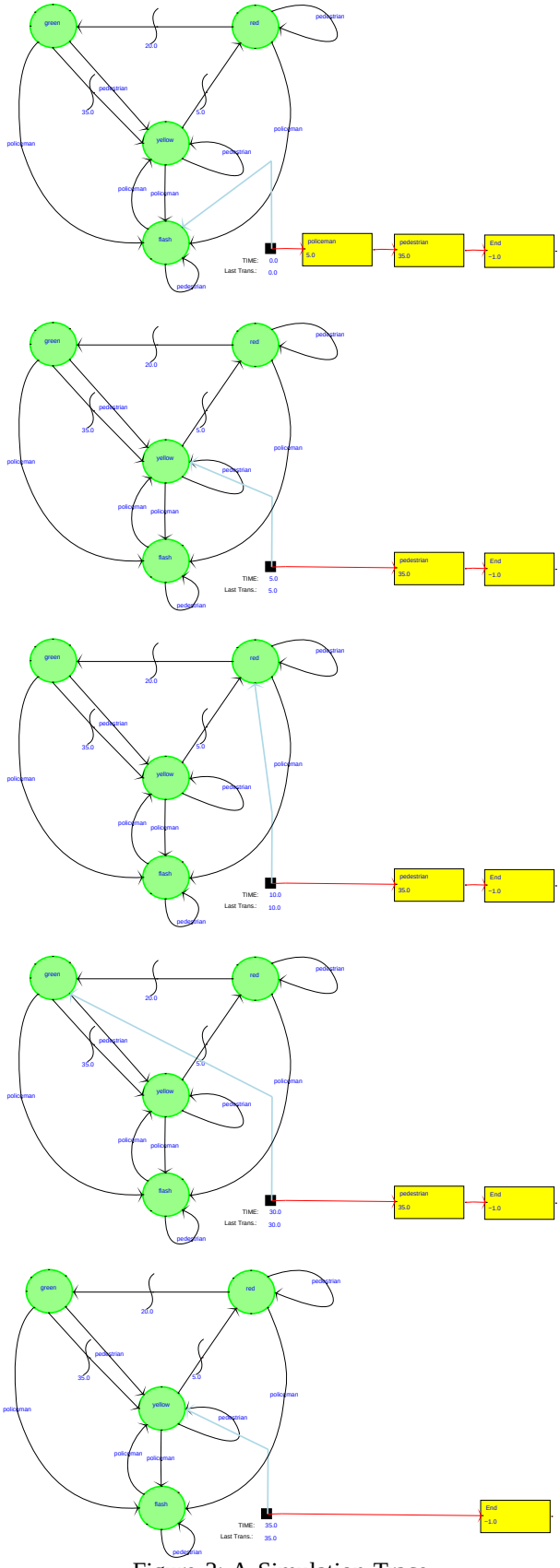


Figure 3: A Simulation Trace

certain state or whether the number of tokens (which might represent resources in a system) remains bounded. The generated Timed Transition Petri Net of the Traffic Light in Figure 1 is shown in the small window in the lower left corner, pointed to by the arrow *Transformation*.

Graph grammars for formalism transformation are particularly useful for the modelling and analysis of complex systems. Models of such systems consist of many components or views, possibly at different levels of abstraction. Due to the diversity of these models, we use different formalisms to describe each one of them. To analyse the entire system, one cannot look at properties of components or views in isolation, but the system should be understood as a whole. Therefore, in *Computer Automated Multi-Paradigm Modelling* (Mosterman and Vangheluwe 2002, Vangheluwe et al. 2002) modelling and simulation, we have proposed to translate each component or view into a single common formalism for subsequent analysis and simulation. We have previously used these ideas to model, simulate and analyse complex physical as well as logical systems. In de Lara et al. (2003a) and de Lara et al. (2003b), we focus on hybrid systems design; in Posse et al. (2002) and Mosterman and Vangheluwe (2000), we focus on continuous models; in de Lara and Vangheluwe (2002c), AToM^3 is used as a meta-case tool; in de Lara and Vangheluwe (2002d) and Posse and Bolduc (2003), the visual modelling and simulation environments for respectively the discrete-event modelling formalisms GPSS and DEVS are constructed; and in de Lara and Vangheluwe (2002b), the transformation of Statecharts onto behaviourally equivalent Petri Nets is described.

3.3 Code Generation

The State Automaton model of the traffic light behaviour in Figure 1 is constructed at a suitably high abstraction level and in the appropriate formalism. It fosters understanding, maintenance, and reuse. Transformation to the Timed Transition Petri Net formalism and subsequent analysis gives insight into classes of behaviour and pertinent behavioural features such as termination and deadlock. Simulation of the model gives detailed insight into the system's reactive behaviour, given a specific initial state and series of input events. The ultimate goal of the modelling effort remains however the production of reliable and efficient executable code. Hence, a code generation transformation produces Python code from the Timed Automata model. For demonstration purposes (as opposed to production-code to be embedded in appropriate traffic light controller hardware), this Python code implements a GUI with buttons to let a user produce the events that may occur in the model: *policeman* and *pedestrian*.

We have manually coded classes specifying the static setup of the GUI: the varous widgets (from the Tkinter package) and their layout. It is noted that this part of

the application would usually be modelled in UML Class Diagrams or even more intuitively in a visual editor allowing the user to interactively place widgets. In either case, the framework only takes care of static aspects. This must be complemented with code generated from our Timed Automata model of the system's dynamics.

To match the Tkinter binding mechanism, generated methods correspond to *events* raised by user clicks on the *policeman* and *pedestrian* buttons. Other approaches are possible as described in Horrocks (1999). To simplify the specification of widget methods to be invoked from the state automaton, the latter has been augmented with entry actions. In particular, entering a state will change the traffic light's colour in the appropriate canvas widget by calling the hand-coded `show` method.

The graph grammar modelling the code generation is not presented here as it is trivial. Rather, the structure of the generated code is given.

Each of the states found in the model is encoded as an integer for efficiency reasons:

```
self.red=0
self.yellow=1
self.green=2
self.flash=3
```

We use Tkinter's event-scheduling capabilities to schedule the future invocation of the `timeout` callback. The model gives the system's initial state. This allows us to set the current `state`.

```
self.state=self.red
# entry action for initial state
self.show(self.state)
# schedule timeout from initial state
self.schedule()
```

Our model contains a list of scheduled input events. This information will be ignored for code generation as in the application, these events are externally generated by the user. The `schedule` method checks the current state and uses Tkinter's `after` to schedule the invocation of the `timeout` method after the number of milliseconds (specified in the model in seconds).

```
def schedule(self):
  if (0):
    pass
  elif (self.state == self.red):
    self.timeout_ref=self.after(20000, self.timeout)
  elif (self.state == self.yellow):
    self.timeout_ref=self.after(5000, self.timeout)
  elif (self.state == self.green):
    self.timeout_ref=self.after(35000, self.timeout)
```

The `timeout` method checks the old state, executes the new state's entry method `show()`, sets the current state to the new state, and schedules the timeout event for this new state. The new state is obviously found in the model.

```
def timeout(self):
  if (0):
    pass
  elif (self.state == self.red):
    self.show(self.green)
    self.state=self.green
    self.schedule()
  elif (self.state == self.yellow):
    self.show(self.red)
    self.state=self.red
    self.schedule()
  ...
```

Handling external interrupts such as `policeman` is completely analogous.

```
def policeman(self):
  if (0):
    pass
  elif (self.state == self.red):
    self.show(self.flash)
    self.state=self.flash
    self.schedule()
  elif (self.state == self.yellow):
    self.show(self.flash)
    self.state=self.flash
    self.schedule()
  ...
```

The application generated from the model of the traffic light is shown in the small window at the bottom of Figure 1, pointed to by the *Code Gen.* arrow. The generated class TrafficGUI was instantiated three times to demonstrate how the code is correctly encapsulated correctly implementing three concurrent, independent traffic lights.

4 RELATED WORK

Several tools similar to AToM^3 exist in the graph grammars community. Examples are GenGed (Bardohl 2002) and DiaGen (Minas 2002). In the latter, the user gives a textual specification of the visual language and obtains a set of Java classes which are complemented by a Java library to obtain the visual environment. In AToM^3, the specification of the visual language (the meta-model) is done graphically, and the generated files are loaded by the AToM^3 kernel. There is no structural difference between the generated editors (which could be used to generate other ones), and the editor which generated them. In fact, one of the main differences of the approach taken in AToM^3 with other similar tools is the concept that (almost) everything in AToM^3 has been defined by a model (under the rules of some appropriate formalism) and can thus be modified by the user.

In the meta-modelling community, tools such as DoME (Honeywell 1999) or MetaEdit+ (Phjonen and Tolvanen 2002) use a textual, low-level language (Alter in the case

of DoME) for the definition of the model manipulations. In contrast, in our approach, the user can define transformations as models in the graph grammars formalism. Among other advantages (section 3), this frees the user from the need to know too many details of the internals of the tool.

With the growing importance of the OMG's Model Driven Architecture (MDA), tools such as Codagen's (`www.codagen.com`) Architect support the explicit separation of Platform Independent Models (PIM) and Platform Specific Models (PSM). The specification of code generation explicitly describes mapping of UML models onto code. This specification is reusable, though not a model in its own right.

5 CONCLUSIONS

In this paper, we have presented a framework for model-based development founded on a combination of meta-modelling and graph transformation. By means of meta-modelling we graphically specify the syntax of models we want to deal with. By means of graph transformation we graphically and rigourously define the kinds of manipulations that we can apply to these models. These manipulations typically include defining operational semantics, transformations into other formalisms, code generation and optimization. Using graph transformation has the advantage that computations are explicitely modelled. We have implemented these concepts in the *Multi-Paradigm* tool AToM^3 following the *everything is a model* philosophy.

By example, we have shown the definition of the Timed Automata formalism and of model manipulations to simulate, transform into Petri Nets and generate code. This has been applied to a simple model of a traffic light.

Summarizing, our approach to automating code generation consists of three steps. The first is to define an appropriate visual language powerful enough to represent the variability of the family of applications. The users will use this formalism to model the application they want to produce. The second step is to code a framework of classes (implementing the parts of the application that do not change) that will be complemented by the code generated from the model of the application which the user graphically specified. Finally, the third step requires the specification of a code generator for the model. In our approach this transformer too was explicitly modelled using the graph grammar formalism.

In the future, we will enhance the flexibility of the tool, by for example extending the scope of the *Buttons* formalism in which models of the user interface are generated. The automatically generated user interface models will be able to control more complex behaviours of the tool. We are also working on the core of meta-modelling and generalizing common aspects that can be applied to many formalisms. Among others, these aspects include inheritance, hierarchy and ports. We are also improving the power of our graph grammar engine allowing more complex pattern-matchings, for example allowing negative application conditions and matching of structural subtypes of the nodes in the LHS (which are found at run-time, that is, the subtyping does not need to be declared in the meta-model). Finally, we are applying these concepts to the verification of UML models, in the modelling and analysis of complex physical systems and for educational purposes in several doctoral and graduate courses on advanced modelling and simulation based design.

ACKNOWLEDGMENTS

Juan de Lara's work has been partially sponsored by the Spanish Interdepartmental Commission of Science and Technology (CICYT), project number TEL1999-0181. Hans Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant.

REFERENCES

Bardohl, R. 2002. A visual environment for visual languages. *Science of Computer Programming* 44:181–203. See also the GENGED home page: `http://tfs.cs.tu-berlin.de/~genged/`.

de Lara, J., and H. Vangheluwe. 2002a, April. 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)*, LNCS 2306, 174 – 188: Springer-Verlag. Grenoble, France. See also the AToM^3 home page: `http://atom3.cs.mcgill.ca`.

de Lara, J., and H. Vangheluwe. 2002b, October. Computer aided multi-paradigm modelling to process petri-nets and statecharts. In *International Conference on Graph Transformations (ICGT)*, Volume 2505 of *Lecture Notes in Computer Science*, 239–253: Springer-Verlag. Barcelona, Spain.

de Lara, J., and H. Vangheluwe. 2002c, April. Using AToM^3 as a Meta-CASE tool. In *4th International Conference on Enterprise Information Systems (ICEIS)*, 642 – 649. Ciudad Real, Spain.

de Lara, J., and H. Vangheluwe. 2002d, June. Using meta-modelling and graph grammars to process GPSS models. In *16th European Simulation Multi-conference (ESM)*, ed. H. Meuth, 100–107: Society for Computer Simulation International (SCS). Darmstadt, Germany.

de Lara, J., H. Vangheluwe, and M. Alfonseca. 2002, October. Using meta-modelling and graph grammars to create modelling environments. In *International Conference on Graph Transformations (ICGT), Workshop on Visual Modelling Techniques*, Volume 72 No. 3 of

Electronic Notes in Theoretical Computer Science: Elsevier Science. 15 pages. Barcelona, Spain.

de Lara, J., H. Vangheluwe, and M. Alfonseca. 2003a. Metamodelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSyM)*. 14 pages. (in press).

de Lara, J., H. Vangheluwe, and M. Alfonseca. 2003b, July. Computer Aided Multi-Paradigm Modelling of Hybrid Systems with AToM^3. In *Summer Computer Simulation Conference*: Society for Computer Simulation International (SCS). Montréal, Canada.

Ehrig, H., G. Engels, H.-J. Kreowski, and G. Rozenberg. 1999. *Handbook of graph grammars and computing by graph transformation. vol. 2: Applications, languages, and tools*. World Scientific.

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

Horrocks, I. 1999. *Constructing the user interface with statecharts*. Addison-Wesley.

Minas, M. 2002. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming* 44:157–180. See also the DIAGEN home page: `http://www2.informatik.uni-erlangen.de/DiaGen/`.

Mosterman, P., and H. Vangheluwe. 2000, September. Computer automated multi paradigm modeling in control system design. In *IEEE International Symposium on Computer-Aided Control System Design*, ed. A. Varga, 65–70: IEEE Computer Society Press. Anchorage, Alaska.

Mosterman, P. J., and H. Vangheluwe. 2002. Computer automated multi-paradigm modeling. *ACM Transactions on Modeling and Computer Simulation* 12 (4): 1–7. Special Issue Guest Editorial.

Murata, T. 1989, April. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77 (4): 541–580.

Peterson, J. 1981. *Petri net theory and the modeling of systems*. Prentice Hall.

Phjonen, R., and J.-P. Tolvanen. 2002. Automated production of family members: Lessons learned. In *Proceedings of the Second International Workshop on Product Line Engineering - The Early Steps: Planning, Modeling, and Managing (PLEES'02)*, 49–57.

Posse, E., and J.-S. Bolduc. 2003, July. Generation of DEVS simulators by graph-transformation. In *Summer Computer Simulation Conference. Student Workshop*: Society for Computer Simulation International (SCS). Montréal, Canada.

Posse, E., J. de Lara, and H. Vangheluwe. 2002, April. Processing causal block diagrams with graph-grammars in AToM^3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, 23 – 34. Grenoble, France.

Sztipanovits, J., G. Karsai, C. Biegl, T. Bapty, A. Ledeczi, and A. Misra. 1995, November. MULTIGRAPH: An architecture for model-integrated computing. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, 361–368. Ft. Lauderdale, Florida. See also the GME home page: `http://www.isis.vanderbilt.edu/Projects/gme/`, Vanderbilt University.

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

AUTHOR BIOGRAPHIES

HANS VANGHELUWE is an Assistant Professor in the School of Computer Science at McGill University, Montréal, Canada where he teaches Modelling and Simulation, as well as Software Design. He also heads the Modelling, Simulation and Design Lab (MSDL). He has been the Principal Investigator of a number of research projects on the development of a multi-formalism theory for Modelling and Simulation. Some of this work has led to the WEST++ tool, which was commercialised for use in the design and optimization of Waste Water Treatment Plants. He was the co-founder and coordinator of the European Union's ESPRIT Basic Research Working Group 8467 "Simulation in Europe", and a founding member of the Modelica Design Team. His current research is focused on the development of the AToM^3 tool for Computer Aided Multi-Paradigm Modelling (CAMPaM) and simulation. His e-mail address is `<hv@cs.mcgill.ca>`, and his web page is `<www.cs.mcgill.ca/~hv>`.

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