Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Visual Languages and Computing
15 (2004) 309–330
Journal of Visual Languages & Computing
www.elsevier.com/locate/jvlc

Defining visual notations and their manipulation through meta-modelling and graph transformation

Juan de Lara[a,*], Hans Vangheluwe[b]

[a] *Ing. Informática, Universidad Autónoma de Madrid, Madrid, Spain*
[b] *School of Computer Science, McGill University, Montréal, Canada*

Received 13 June 2003; received in revised form 2 November 2003; accepted 5 January 2004

Abstract

This paper presents a framework for the definition of visual notations (both syntax and semantics) based on meta-modelling and graph transformation. With meta-modelling it is possible to define the syntax of the notations we want to deal with. Meta-modelling tools are able to generate environments which accept models in the defined formalisms. These can be provided with further functionality by defining operations that can be performed to the models. One of the ways of defining such manipulations is through graph grammars, because models and meta-models can be represented as attributed, typed graphs. In this way, computations become high-level models expressed in the formal, graphical and intuitive notation of graph grammars. As an example, AToM^3 is used to automatically generate a tool for a *Discrete Event Simulation* notation. The tool's functionality has been completely defined in a visual way through graph grammars, and includes a simulator (formalism's *operational semantics*), a transformation into Timed Transition Petri nets (*denotational semantics*), an optimizer and a code generator for a GPSS simulator.

Keywords: Meta-modelling; Graph transformation; Domain-specific visual languages; Discrete-event simulation; AToM^3

*Corresponding author.
E-mail addresses: juan.lara@ii.uam.es (J. de Lara), hv@cs.mcgill.ca (H. Vangheluwe).

1045-926X/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2004.01.005

1. Introduction

For as long as computers were invented more than half a century ago, *Software Engineers* have sought higher levels of abstraction to program their applications: from bits to assembler to procedural programming languages and object orientation. This evolution seems natural and necessary for several reasons, among them the increasing needs of productivity and quality. 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 [1].

Domain-specific visual languages (DSVL) are graphical notations specially devised for the specific needs and knowledge of a certain group of users. DSVL have the advantage of being at a very high-level of abstraction and very effective and intuitive for the task to be performed. In areas where DSVL are intensively used or where the notations can evolve, we need a way to reduce the effort in building and maintaining the DSVL tools. Meta-modelling is a way to reduce this problem, as one can use graphical, high-level notations (such as UML class diagrams with OCL constraints) to define the syntax of the visual language. From this meta-model, a tool can be generated for the defined formalism. The generated tool has a limited functionality, typically editing, loading, saving models and verifying that they are correct (consistent with the meta-model). However, for a modelling tool to be useful, more complex operations on the models are desired. For example, simulation, model transformation into some other (textual or graphical) notation (where other operations could be performed, or properties could be proved) and model optimization (for example, reducing its complexity). As models, meta-models and meta–meta-models can be represented as attributed, typed graphs, these operations can be expressed using graph grammars [2]. In this way, computations become high-level models expressed in a formal, graphical and intuitive notation. This eliminates the problem of adding features to the generated tool using a lower-level programming language and has the potential to improve key factors in software development, such as productivity, quality and ease of maintenance.

AToM^3 [3] (A Tool for Multi-Formalism and Meta-Modelling) is a Multi-Paradigm Modelling tool that is being developed in collaboration between McGill University and the Universidad Autónoma in Madrid. This tool implements the concepts explained above and has been used to define modelling environments for areas such as Modelling and Simulation (Petri nets [4], GPSS [5] and Causal Block Diagrams), Artificial Intelligence (Constraint Satisfaction Problems and Ant Colony Optimization) and Software Engineering (the UML structural and behavioural diagrams [6], Data Flow Diagrams and Structure Charts).

In this paper, we show new improvements in the tool, together with an example of the usefulness of the DSVL approach. We define a notation for *Discrete Event Simulation*, in the *Process Interaction* style [7] and the subsequent automatic generation of a modelling tool. The tool's functionality is enriched by graphically defining graph transformations to simulate, transform into Timed Transition Petri nets (TTPN) [8], optimize the model and generate code for a GPSS simulator [9].

The rest of the paper is organized as follows: Section 2 gives some background on graph grammars and the rationale for its use in our context; Section 3 explains the use of AToM^3 for meta-modelling a Process Interaction Notation; Section 4 presents a graph grammar for simulation; Section 5 shows a graph grammar for the transformation into TTPN for analysis; Section 6 presents a simple graph grammar to improve model efficiency; Section 7 explains the graph grammar for GPSS code generation; Section 8 shows related research and finally Section 9 presents the conclusions and the future work.

2. Graph grammars: a brief introduction

Graph grammars [2] are similar to Chomsky grammars [10] (which are applied on strings), but rules have graphs in left- and right-hand sides (LHS and RHS). Graph grammars are useful to generate sets of valid graphs or to specify operations on them. In the latter case, a graph rewriting processor looks for graph matchings between the LHS of a rule and a zone of an input graph (called *host graph*). When this happens, the matching subgraph in the host graph is replaced by the RHS. Rules may have conditions that must be met in order for the rule to be applied and actions that are performed once the rule is applied. Some graph rewriting processors (such as AToM^3) iteratively apply a list of rules (ordered by priority) to the host graph until none of them is applicable. When a rule can be applied, the processor starts again trying the rule at the beginning of the list. Other processors have a (possibly graphical) control language to select the rule to be considered next.

In our approach, we use graph grammars to specify operations on models (typically model execution, optimization and transformation) at any meta-level, as these can be expressed as attributed, typed graphs. In this case, the attributes of the nodes in the LHS must be provided with the matching conditions. In AToM^3, we can specify that either a specific value or any value will make a match. Nodes in both LHS and RHS are also provided with labels to specify the mapping between LHS and RHS. If a node label appears in the LHS of a rule, but not in the RHS, then the node is deleted when the rule is applied. Conversely, if a node label appears in the RHS but not in the LHS, then the node is created when the rule is applied. Finally, if a node label appears both in the LHS and in the RHS of a rule, the node is not deleted. If a node is created or maintained by a rule, we must specify in the RHS the attributes' values after the rule application. In AToM^3 there are several possibilities. If the node label is already present in the LHS, the attribute value can be copied. We also have the option to give it a specific value or assign it a program to calculate the value, possibly using the value of other attributes.

Using a model of the computation in the form of a graph grammar has several advantages over embedding the computation in a lower-level, textual language. Graph grammars are a natural, graphical, formal and high-level formalism. Its theoretical background can help in demonstrating the termination and correctness of the computation model. As a natural and intuitive formalism for describing computations, it may be interesting for education [11], as computations, and their

effects can be visually traced. Nonetheless, its use is constrained by efficiency as 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 in the matching process.

3. Meta-modelling a process interaction notation with AToM^3

This section presents a meta-model for a Discrete Event Modelling formalism [7] for which transformations are described in the following sections. Discrete Event formalisms usually follow an Event-based or a Process Interaction approach. The former focuses on modelling the events (state changes) in the system, while the latter (the one we follow in this example) models the life-cycle of some entities in the system. The notation we present has been tailored to the domain of manufacturing. Models are composed of pieces moving through an interconnected network of machines. The concepts of the notation are *Pieces*, *Machines*, *Queues* and *Generators* (which produce pieces at certain rates). A meta-model built with AToM^3 for this notation is shown in Fig. 1.

In AToM^3, we can use either the Entity-Relationship or the UML Class Diagrams meta-formalisms for meta-modelling. A meta-formalism can be used to define formalisms as well as other meta-formalisms. In addition, meta-models can be provided with textual constraints expressed as OCL [12] (not shown in the paper) or Python code (''*well-formedness rules*''). These are executed at run time and guarantee

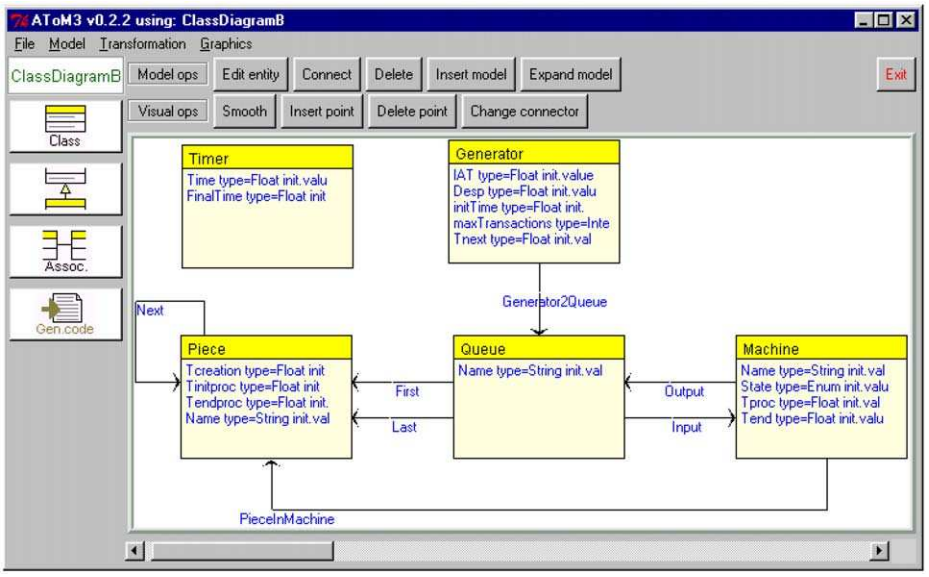

Fig. 1. Meta-model for process interaction.

model correctness. The AToM^3 user interface changes depending on the loaded (meta-)formalism. In the case of Fig. 1, UML class diagrams have been loaded. This notation allows to create *classes*, connect them (by means of inheritance or as a regular connection), or generate code for the described formalism. This code can be loaded again on top of AToM^3, which then allows modelling under the syntax of the defined formalism.

The window in Fig. 1, shows five UML classes for our notation. *Machines* can be either in state *Idle* or *Busy*, consume some time (*Tproc*) for the processing of each *Piece* and have a pointer to the *Piece* that is currently being processed. *Generators* are provided with an inter-arrival time (*IAT*) and a displacement (*Desp*), in such a way that the next *Piece* will be generated in a random number of time units in the interval [IAT-Desp, IAT + Desp]. *Generators* produce a maximum of *maxTransactions Pieces* (if this attribute is greater than zero), starting at time *initTime*. *Generators* can only be connected to *Queues*, the pieces they generate are stored in them. *Queues* have connections to the first and last *Piece* they are storing. A *Piece* can be related to the next *Piece* in the *Queue*. The *Timer* accounts for the final and actual time of the simulation.

In addition to this abstract syntax information, a graphical appearance has been defined for each class and connection in the meta-model. In AToM^3, there are two kinds of graphical attributes: arrow-like and icon-like. The first one is usually assigned to connections in the meta-model, the second one is usually assigned to entities. A graphical editor allows designing both kinds of graphical appearances.

Fig. 2 shows AToM^3 once loaded with the automatically generated files from the previous meta-model. The column of buttons to the left has changed with respect to the window in Fig. 1. This part of the user interface is indeed a model that AToM^3 generates from the meta-model. This model has a meta-model (“*Buttons*”, which is made of a single entity, the *button*) on its own right, and gets interpreted when AToM^3 loads the corresponding formalism. AToM^3 generates a button for each non-abstract UML class in the meta-model. This process is indeed a formalism transformation: from UML class diagrams to the “*Buttons*” formalism. Like some other built-in features of the AToM^3 kernel, it has been modeled using a graph grammar. As the user interface is a model, we can modify it. In the case of the figure we have added buttons to simulate, convert to TTPN, optimize and generate GPSS code (the four lower buttons in the left column). These buttons call the appropriate graph grammar models to perform the operations and are described in the following sections.

Thus, in the *pure meta-modelling* approach we follow, the Visual Language (VL) is completely defined by a meta-model. This can be regarded as a type graph (with inheritance) enriched with certain constraints. Some of them are visually embedded (cardinalities) in the type graph, while others have been defined with a textual language (“*well-formedness*” rules). On the other hand, in a *pure graph grammar* approach [13], the VL is defined by constructing graph grammar rules for generation or parsing (some systems automatically derive rules for generating the alphabet symbols). This usually is more difficult, but the theoretical results of graph transformation can be used to prove properties of the generated language. While the

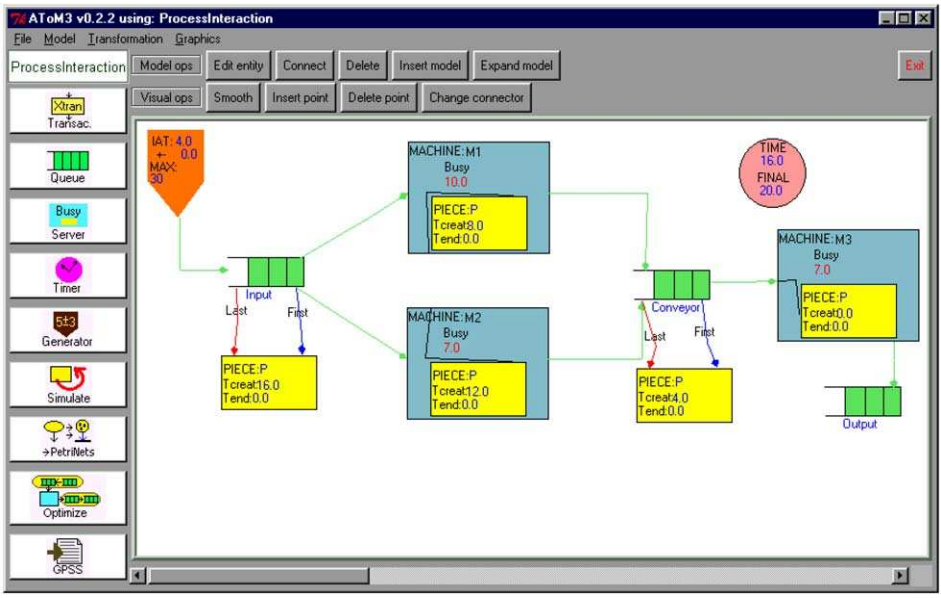

Fig. 2. An example process interaction model.

pure meta-modelling approach is more declarative, the *pure graph grammar* approach is more constructive.

4. Defining a simulator

In this section, we extend the functionality of the tool with a simulator modeled using graph grammars. Coding the simulator in a textual language (Python in the case of AToM^3) can be more efficient, but, for our purposes, graph grammars have a number of advantages (besides the ones expressed in Section 2). Graph grammars should be considered as a notation for high-level modelling in opposition to programming. Thus, a simulator expressed with graph grammars may be viewed as a reference (and executable) specification from which more efficient simulators can be derived. They also have advantages from the point of view of education. Modelling a simulator with graph grammars is natural and intuitive, and one gets insight in the process one is modelling. Additionally, AToM^3 is able to animate and execute graph grammars step by step. In this case, the simulation can be visually traced without having to code complex graphical routines. In AToM^3 one can also use Python to express the computations, but in this case, the user has to know the Python syntax and some of the internal details of the tool, such as the way in which models are stored in memory, and how nodes are connected.

The graph grammar for simulation is composed of six rules. Fig. 3 shows the first two of them. They are numbered according to their priority. Both deal with the production of pieces by a *Generator* and are only applicable if the actual simulation

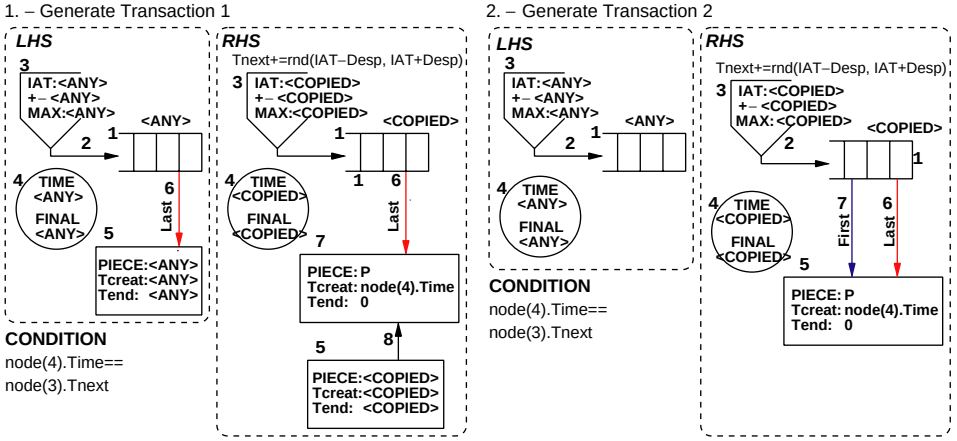

Fig. 3. The first two rules for the simulation graph grammar: generating pieces.

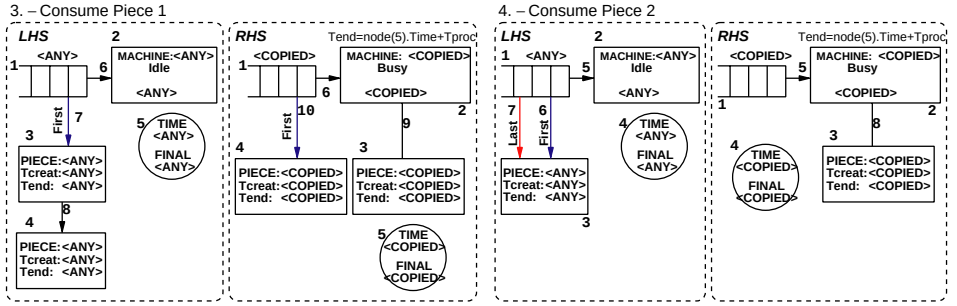

Fig. 4. Rules three and four for the simulation graph grammar: begin of machine processes.

time is equal to the time in which the next *Piece* should be generated (see their conditions). Rule 1 deals with the case in which there are already some *Pieces* in the *Queue* (label "1") attached to the *Generator* (label "3"). The last *Piece* is pointed by the '*last*" arrow. In this case, the newly generated *Piece* (labelled as "7" in the RHS) becomes the last one. Rule 2 shows the special situation in which the *Queue* is empty. In this case, the newly created *Piece* (labelled as "5") is pointed by the "*first*" and "*last*" arrows of the *Queue*. Note how, in the RHS some of the nodes' attributes are copied, while others are assigned new values. This is the case with the *Generator*'s attribute *Tnext*, which stores the time in which the next *Piece* has to be generated. The creation time of the piece (*Tcreat* attribute in nodes 7 and 5 in the RHS) is also calculated.

Fig. 4 shows rules number three and four, which deal with the consumption of *Pieces* by *Machines*. Rule 3 shows a situation in which the incoming *Queue* has more than one stored *Piece*, while in rule 4, the *Queue* has only one *Piece*. In both cases, the first *Piece* in the *Queue* is consumed by the *Machine* (removed from the *Queue*

and pointed by the *Machines* arrow), and the machine final processing time is updated.

Fig. 5 shows rules 5 and 6. Both deal with the end of processing by *Machines*, in which *Pieces* are stored in the output *Queue*. Rule 5 is a general case, in which the *Queue* is not empty and only the "*last*" arrow should be redirected. In rule 6, the *Queue* is empty, and both the "*first*" and "*last*" arrows should be created.

Finally, rule 7 (not shown in the figure, its LHS and RHS consist of a *Timer*) is applied if none of the previous rules can be applied and updates the simulation time with the time of the next event. This is the minimum of the final processing time of each *Machine*, and of the next *Piece* production in each *Generator*. We may have several applications of the previous rules (1–6) before an increment in the simulation time takes place. The applicability condition of rule 7 checks that the actual time must be less than the final simulation time, otherwise the rule is not applicable and the graph grammar execution ends.

Fig. 6 shows the execution of some rules (until a time increment) on a model composed of two machines connected sequentially by *Queues*. During the simulation, it can be the case that a rule can be applied in several zones of the model. For example, if both *Machines* are in the *Idle* state and have a *Piece* in the incoming *Queue*, rule number 4 ("*Consume Piece-2*") can be applied in two different places. In this case, AToM^3's Graph Rewriting module offers several possibilities. In the first one, the system asks the user in which zone in the graph the rule should be applied by clicking in the different subgraphs (in the example, the user should indicate which *Machine* consumes the *Piece*). In the second possibility AToM^3 chooses a random occurrence of the rule. Finally, if the zones in the graph where the rule can be applied are disjoint, AToM^3 can apply the rule in parallel (that is, in the example, both machines would consume the pieces at the same time). For some cases, this non-determinism is not important in the present graph grammar. The time update rule is the last one in the list and we can apply several rules at the same simulation time. That is, in the example, and choosing whichever possibility, both *Pieces* are consumed by both machines at the same simulation time, so the simulation result is the same in any case. In the case of an idle *Machine* with two non-empty input *Queues*, either the user or AToM^3 at random chooses the *Queue* from which the *Piece* comes, and the simulation results of the two alternatives are not the same.

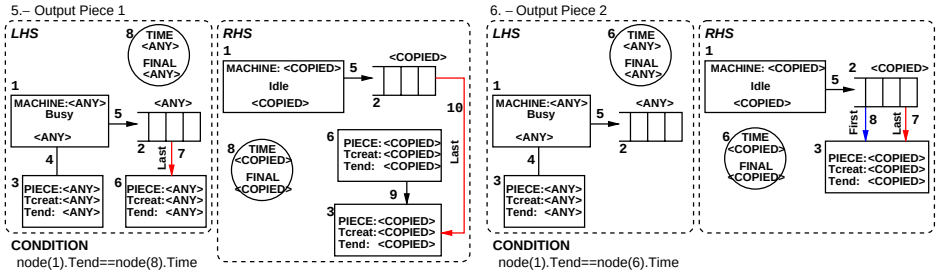

Fig. 5. Rules five and six for the simulation graph grammar: end of machine process.

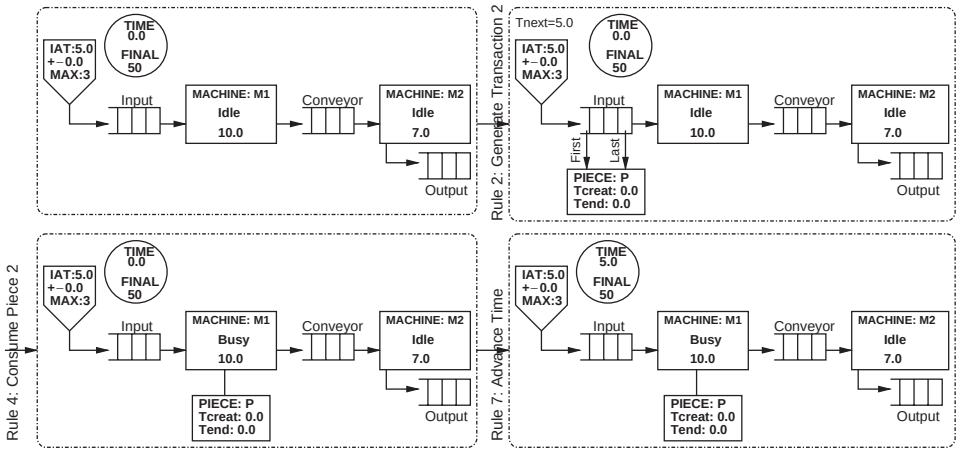

Fig. 6. Some steps in the execution of a simple model.

AToM^3 allows several execution modes for a graph grammar: continuous (only the final model is shown), step-by-step (the user clicks on a button after the execution of each rule) and animated. In this case, one can set the time interval that must be spent after the execution of each rule. The time is an attribute of each rule and is established when the user builds the graph grammar, but it can be changed by the rule. For this kind of graph grammar (simulation grammars), the feature is very useful, because animation can be synchronized with the simulation. For example, if rule number 7 advances time in 5 time units, the rule can set the delay parameter for animation to 5 s.

Another new feature included in the AToM^3's graph grammar engine, is that in the matching process, we can specify either an exact-type matching between the nodes of the LHS and the nodes in the host graph or what we call a "*subtype matching*". In the latter case, nodes (or connections) in the LHS and in the host graph do not need to have exactly the same type. AToM^3 checks at run-time whether the node (or the connection) in the host graph has at least the same set of attributes (and connections) 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, but this relationship is found at run-time. This idea is extremely powerful as one can write general graph grammars and reuse them for many formalisms, in unexpected situations. For example, suppose we want to adapt the formalism defined in this section introducing different kinds of *Pieces* and *Machines* with extra attributes. This could be done by subclassifying the *Piece* and *Machine* classes in the meta-model of Fig. 1. In spite of the modifications performed to the meta-model, the graph grammar for simulation would still be valid. The newly created elements have the same subset of attributes as the ones that appear in the graph grammar rules and these could make a match with the new kinds of *Pieces* and *Machines*. The graph grammar would also be valid in the case we change the application domain, for example, people waiting for cashiers in a supermarket.

5. Transforming into timed transition Petri nets

For the study of certain characteristics of the model, it may be useful to translate it into another formalism for which there are appropriate analysis or simulation techniques. The transformation should preserve the properties under investigation, and can be regarded as expressing the *denotational semantics* of a formalism in terms of another one. This idea was represented in a "*formalism transformation graph*" (FTG) in [14]. That work shows several continuous and discrete event simulation formalisms as the nodes in the graph, while behaviour-preserving transformations are represented as the edges. For the simulation of a complex system (with interconnected components, described in different formalisms) one could transform each of its components into a common formalism and then simulate the system as a whole. The common formalism can be found by looking in the FTG. The same approach is also valid if the goal is not simulation but verification of a property. This approach contrasts with *co-simulation* [15], where no transformation into a common formalism is performed. In that approach, a simulator for each component is needed, and they synchronize by using a coordinator to direct the events appropriately. Thus, the *co-simulation* approach cannot be used to verify a property for which we have to understand the system as a whole, as there is no global, unified view of the system.

In AToM^3, a part of the FTG has been implemented, where formalisms (the nodes in the graph) have been meta-modeled and the edges in the FTG (which depict transformation, simulation and optimization) have been formally expressed as graph grammars [3]. Here, we show a graph grammar for the transformation of models in the Process Interaction formalism described in the previous section into TTPN [28]. For simplicity and space constraints, we transform Process Interaction models in their initial state, that is, when the simulation has not begun, before the generation of any *Piece*. Transitions in TTPN are given a delay for firing since they are enabled. The analysis and simulation methods available for TTPN allow us to analyze if certain states are reachable, and to obtain performance metrics using a similar idea to the *Reachability Graph* for untimed Petri nets [16].

In AToM^3, the user may open several meta-models at the same time to define a transformation graph grammar. During the transformation, the model is a mixing of both source and target formalisms, but when the transformation ends, the model is expressed in the target formalism alone. During the transformation process the syntactic constraints imposed by the source and target formalisms are not checked. However, at the end of the transformation, AToM^3 checks that the transformed model is consistent with the syntactic rules of the target meta-model.

AToM^3 allows the application of a list of graph grammars to a model. Decomposing a graph grammar in blocks makes the transformation more reusable, easier to understand and more efficient. This is due to the fact that once a graph grammar in the list is finished, their rules are not considered again by the graph rewriting processor. Graph grammars may have actions (Python code) to be performed before and after the graph grammar execution.

In the example of this section, the transformation can be clearly divided in three graph grammars. The first one is shown in Fig. 7 and attaches Petri net *places* and *transitions*

to each block in the Process Interaction model. The initial action adds attribute *done* (initialized to zero) to all *Generators*, *Queues* and *Machines*. This attribute is deleted by the graph grammar final action. Rule 1 is applied to *Generators* not processed before (*done* attribute equal to zero), and attaches them to a Petri net *transition*. The delay of this transition is set to a function returning a random number in the limits specified in the *Generator*. In this way, the transition fires at the same intervals in which the *Generator* produces a *Piece*. If the rule is applied, the *done* attribute is set to one to prevent the *Generator* from being processed again. Rule 2 attaches a Petri net *place* to each *Queue*. Rule 3 attaches two Petri net *places* to each *Machine*, representing states *Idle* and *Busy*. The rule puts a token in the *place* representing the *Idle* state. Later we will make sure that the number of tokens in both *places* is exactly one, using a well-known structural property of Petri nets for capacity constraint [4].

The second graph grammar is used to connect the Petri net elements according to the connectivity of the attached Process Interaction elements. It is composed of three rules and is shown in Fig. 8. Rule 1 connects the *transition* attached to each *Generator* with the *place* attached to the connected *Queues*. This rule is applied once for each *Queue* attached to each *Generator*. The connection between the *Generator* and the *Queue* is deleted to avoid multiple processing of the same *Generator* and *Queue*.

Rule 2 connects (through a *transition*) the associated *place* of an incoming *Queue* to a *Machine* with the *places* associated with it. When the transition fires, it changes the *Machine* state to *Busy* and removes a token from the *place* representing the *Queue*. If the rule is executed, then the *Queue* is also disconnected from the *Machine* to avoid multiple processing of the same *Queue* and *Machine*. A similar situation (but for output *Queues*) is described in rule number three.

Finally, the last graph grammar (not shown in the figures) removes the Process Interaction blocks. The rules simply disconnect and remove the Process Interaction elements.

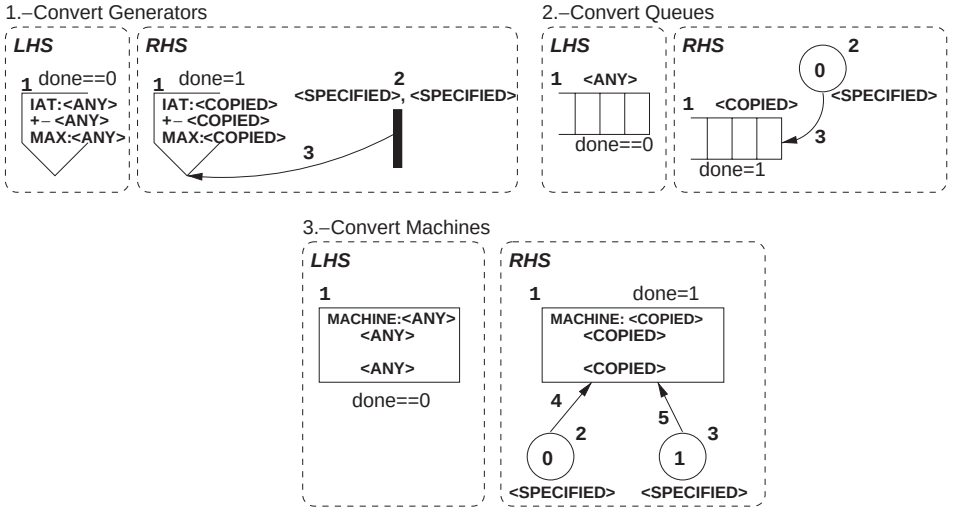

Fig. 7. First block of rules for the transformation graph grammar: adding the petri net elements.

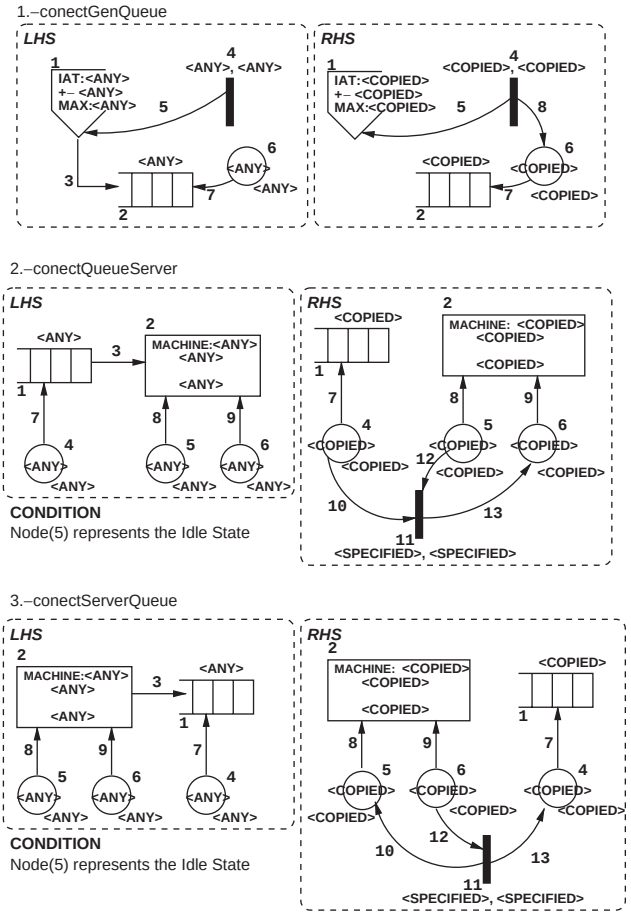

Fig. 8. Second block of rules for the transformation graph grammar: connecting petri net elements.

As an example, Fig. 9 shows the result of the transformation of the Process Interaction model shown in Fig. 2 (without any of the *Pieces*, only the structure is transformed). The sequence of graph grammar rules that where applied for the transformation has been the following: *convertGenerators*, *convertQueues* (3 times), *convertMachines* (3 times), *ConnectGenQueue*, *ConnectQueueServer* (3 times), *ConnectServerQueue* (3 times), *removeQueue* (3 times), *removeMachine* (3 times), *removeGenerator* and *removeTimer*.

6. Optimization

Optimizing transformations do not change the formalism in which the model is expressed, but perform a structural change in the model, which results in a reduction of its complexity, or in an improvement of its performance.

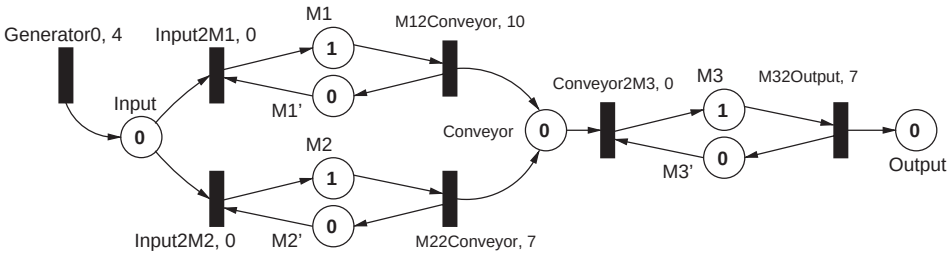

Fig. 9. Result of the transformation of the process interaction model in Fig. 2 into TTPN.

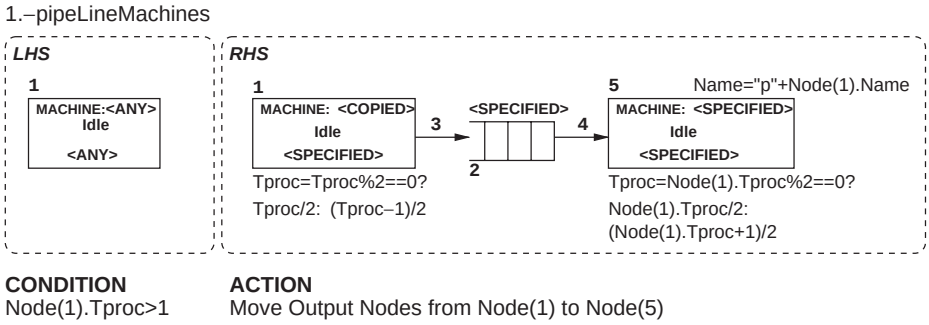

Fig. 10. Graph grammar for optimization: pipelining.

Fig. 10 shows a simple graph grammar for improving the performance of the Process Interaction models. We assume that the process performed by the *Machine* can be decomposed (*pipelined*) in processes up to the time unit. *Pipelining* [17] is a standard procedure to improve the performance of the operations performed by processors. Its use is very common to build fast CPUs. The idea is to divide the process in subprocesses, in such a way that a job passes through the chain of subprocesses and is not finished until it has passed through all of them. All these subprocesses can be active in parallel, thus performing the different work stages (on different jobs) at the same time. In our Process Interaction notation, this means that several *Pieces* can be processed in parallel by the different *Machines* in which we have divided the original *Machine*. This results in an improvement in the rate at which the *Machines* produce the *Pieces* (if more than one *Piece* is waiting in the input *Queue* to be processed).

Thus, the only rule in the graph grammar shown in Fig. 10 takes a *Machine* with a processing time larger than one and generates two *Machines* connected by a *Queue*. As an action after the rule execution, the outputs of the original machine are moved to the newly created one.

7. Code generation

In this section we define a code generator for GPSS (General Purpose Simulation System) [5], which is the basis for the majority of Process Interaction tools available

in the market. It is a block-based textual language, but since it was invented in the fifties, a graphical notation was devised for it. Although all current simulators accept the textual notation, only some of them accept the graphical one. We focus on generating the textual representation which then can be fed into many different simulators. The total number of blocks of the GPSS language varies depending on the specific implementation, but usually there are about 20. For this example we only need some of them, which are explained next:

- "*GENERATE* $\langle IAT \rangle$, $\langle DESP \rangle$" generates *transactions* (*Pieces* in our example), at random in the interval [IAT-DESP, IAT + DESP].
- "*QUEUE* $\langle q \rangle$" and "*DEPART* $\langle q \rangle$" queue and dequeue transactions in the queue named "q".
- "*SEIZE* $\langle x \rangle$" and "*RELEASE* $\langle x \rangle$" take and release resource named "x" for exclusive use. In our Process Interaction notation, resources are *Machines*.
- "*ADVANCE* $\langle m \rangle$, $\langle d \rangle$" which simulates the processing of a transaction during a random time in the interval [m-d, m + d]. We use this block to indicate the time it takes a *Machine* to process a *Piece*.
- "*TRANSFER BOTH*, $\langle x \rangle$, $\langle y \rangle$" is a kind of "go-to" that checks whether a transaction can enter in the block specified by operand $\langle x \rangle$, and in this case, the transaction is sent to it. Otherwise, it is sent to the block specified by label $\langle y \rangle$. This block can also be used as an unconditional transfer if the first parameter is omitted and only one label is given as the second parameter.
- "*TERMINATE n*" counts *n* transactions as terminated (usually 1 or 0).
- "*END*" signals the end of the program code.
- "*SIMULATE* $\langle n \rangle$" runs the simulation until *n* transactions finish.

Code generation is performed graphically by means of graph grammars. Of course, in AToM^3 one can write efficient Python code for this purpose, but by using a graph grammar the user does not have to care about how the model is stored inside AToM^3 and of other implementation details. The code generator becomes another high-level model, that we can build graphically, although in this case, we have to specify the textual patterns that have to be generated as a result of a rule application.

The main idea is to serialize the graphical model, performing a depth-first traversal of the graph. This is performed by using a '*pointer*" that signals to the block whose code should be generated. The meta-model has to be extended to include this *pointer* which can be connected to either *Generators*, *Queues* or *Machines*. Additionally, two kinds of connections are allowed from the pointer to the blocks: the ones that appear as continuous black arrows point to the current block, while the dotted arrows point to a path that has not been generated and should be completed.

The graph grammar for GPSS code generation has an initial action which labels each node in the model with the attribute *done*, and initializes it to zero. This attribute controls whether code has been already generated for the node. This initial action also opens a file ("*f*"), where the textual code is written. Fig. 11 shows the first two rules. Rule 1 generates code for *Generators*. It is applicable if there is no *pointer* in the model

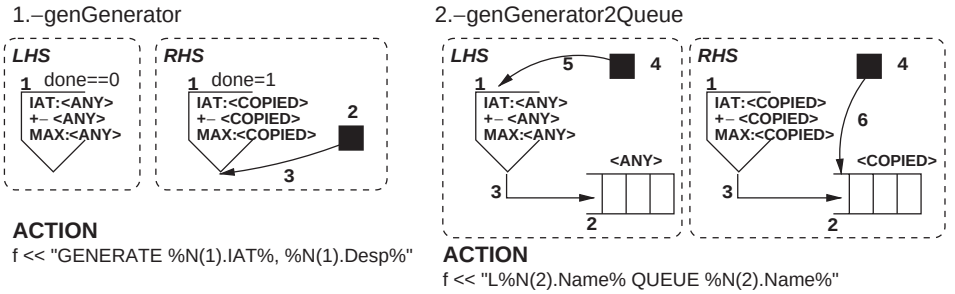

Fig. 11. Rules 1 and 2 for GPSS Code Generation: Generators and Queues.

and the *Generator* has not been processed before. If these conditions are met a *pointer* is created, which points to the *Generator*, and textual code is written in the file. For clarity, we put between the percentage symbols the node attributes whose value is written in the file, and use the notation "$N(x)$" to refer to the node whose label is "x". Rule 2 is applied when there is a *Generator* (the last generated node) connected to a *Queue*. In this case, GPSS code is written to queue a transaction (preceded by an appropriate label) and the *pointer* advances to the *Queue*.

Fig. 12 shows rules 3 and 4, which deal with the beginning of processing of the *Pieces*. Rule 3 shows a situation where a *Queue* is connected to multiple *Machines*. The *Piece* is thus processed by one of the *Idle Machines*. This rule advances the *pointer* to one of the non-processed *Machines*, but keeps a pointer to the *Queue* as other paths in the graph should be processed later. The first line of GPSS code checks whether the *transaction* can reach the first *Machines* (if this is *Idle*), or whether other *Machine* should be tried (if the first one was *Busy*). The code in the second line seizes the first *Machine*, which enters in state *Busy*. Finally, the code in the third line dequeues the *Piece*. A *Piece* only reaches this point if it has been able to seize the resource (the *Machine*). Rule 4 is applied if the *Queue* is only connected to one *Machine*, or only one non-processed *Machine* remains. Due to the priority order of the rules, this rule is applied only if the previous one could not be applied.

Fig. 13 shows the three remaining rules of the graph grammar. Rule 5 deals with the case of a *Machine* (which is the last generated block) connected to an output *Queue*. In this case GPSS code is generated for the processing time, as well as to release the *Machine* (placing it in *Idle* state again). Rule 6 is used when a path in the graph has been completely generated, and other paths should be completed. This happens when the pointer cannot further advance. In this case, the rule simply sets the current *pointer* to one of the non-generated paths. Finally, rule 7 terminates the code generation procedure for one of the connected graphs. The graph grammar has a final action which writes a *SIMULATE* sentence according to the number of *Transactions* that must be generated and an *END* sentence indicating the end of the listing and closes the file.

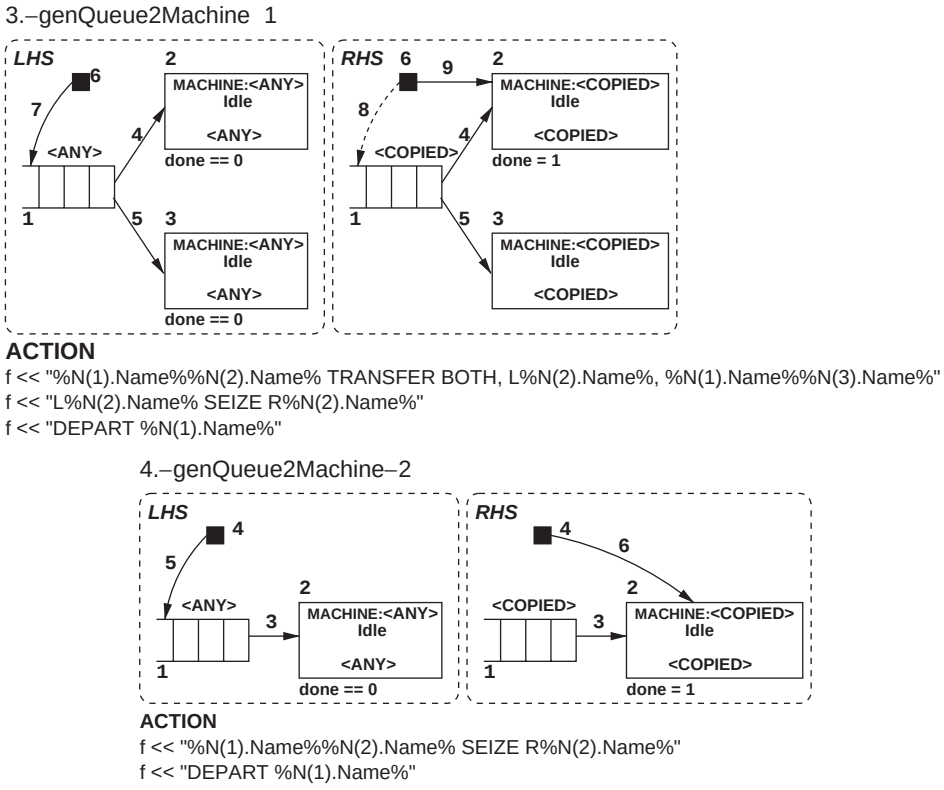

Fig. 12. Rules 3 and 4 for GPSS code generation: begin of process.

The GPSS code generated by the graph grammar from the model shown in Fig. 2 (without the *Pieces*) is shown in listing 1.

```
[1]                GENERATE 4,0            [12]            ADVANCE 7,0
[2]   LInput       QUEUE Input             [13]            RELEASE RM3
[3]   InputM1      TRANSFER    BOTH,L-     [14]            TRANSFER ,LOutput
                   M1,InputM2
[4]   LM1          SEIZE RM1               [15]  LOutput   TERMINATE 1
[5]                DEPART Input            [16]  InputM2   SEIZE RM2
[6]                ADVANCE 10,0            [17]            DEPART Input
[7]                RELEASE RM1             [18]            ADVANCE 7,0
[8]                TRANSFER ,LConveyor     [19]            RELEASE RM2
[9]   LConveyor    QUEUE Conveyor          [20]            TRANSFER ,LConveyor
[10]  ConveyorM3   SEIZE RM3               [21]            SIMULATE 30
[11]               DEPART Conveyor         [22]            END
```

Listing 1: Generated GPSS code from the Process Interaction model in Fig. 2.

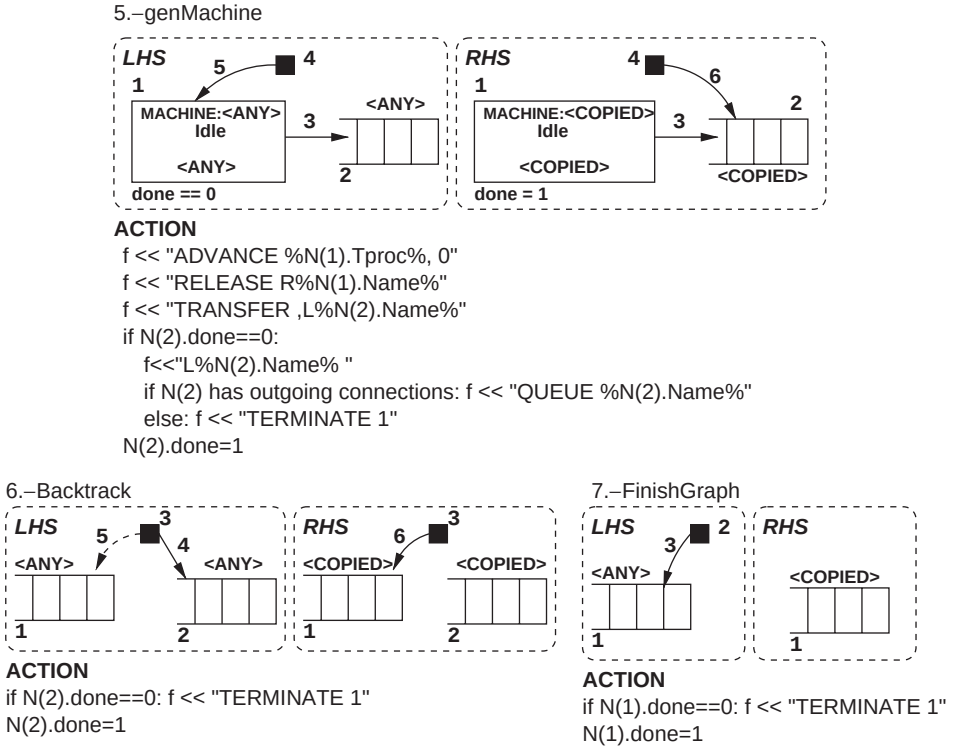

Fig. 13. Rules 5, 6 and 7 for GPSS code generation.

GPSS simulators [9] usually give a standard report showing information about resources (*Machines* in our case) and *Queues*. With respect to the resources, we can have information about the number of transactions processed by the resource, the percentage of time the resource has been used (*utilization*) and the average time the resource was busy. With respect to *Queues*, we can obtain information about the maximum number of stored transactions during the simulation, the total number of them that passed through, the average contents and the average time a transaction spent in the queue. Additionally, we can have information about the number of transactions that passed through each block of the model.

For the experiment performed, the simulation stopped at time 224. The experiment shows that *Machine* M1 had a utilization time of 98.2%, *Machine* M2 of 96.0% and *Machine* M3 of 93.8%. With respect to the *Queues*, the *Input Queue* had a maximum of 3 *Pieces* (0.978 in average) and the *Pieces* waited 3.911 units of time in average. The *Conveyor Queue* had 22 *Pieces* as its maximum number, and 10.040 in average, the *Pieces* had to wait for 43.288 unit of time in average. That is, by means of simulation we can show that this configuration is not stable, as the number of *Pieces* stored in both the *Conveyor* and the *Input Queues* grows without limit as the simulation time increases. Both problems can

be eliminated in many ways, for example, if we reduce the rate at which *Pieces* are generated from 4 to 5 and increase the efficiency of *Machine* M3 from 7 to 5 time units.

8. Related work

In the graph grammar community there are other similar tools, most of them use a *pure graph grammars approach*. For example, GENGED [13] is a tool built on top of the graph transformation tool AGG [18] which supports the definition of VLs. Model correctness is guaranteed by defining creation or parsing graph grammars. On the contrary, in ATOM^3 (*pure meta-modelling approach*) we rely on the evaluation of constraints (defined in the meta-model) when the user builds the model. In our experience, the definition of constraints usually requires less effort than building a creation or parsing graph grammar, but, as stated before, the *pure graph grammar* approach can take advantage of the theoretical foundations of graph transformations for proving properties of the generated languages. Additionally, meta-modelling techniques are heavily used in the context of UML and the Model Driven Architecture (MDA) [19]. Model transformation plays a central role for the later technology, and we believe that the combination of meta-modelling and graph transformation can be a very suitable approach.

Although in GENGED graph grammars can be defined for simulation and animation [20], we also deal with other kinds of manipulations, such as transformation into other formalisms, code generation and optimization. In particular, for the specification of transformations between formalisms, we allow the use of several meta-models at the same time. GENGED uses a constraint language to specify the graphical layout, while we associate constraints (which may have lateral effects, such as performing graphical layouts) with events in a similar way as event programming languages such as Visual Basic. This is lower-level, but usually more efficient. The mapping from abstract to concrete syntax is very simple in ATOM^3, as each abstract syntax entity or connection should have an associated concrete syntax symbol (icon or arrow-like). In GENGED this mapping can be more sophisticated. Thus, the approach of ATOM^3 makes easy the definition of graph-like VLs, while arbitrary languages with complicated layouts (such as Nassi–Schneiderman diagrams for example) are much more difficult to describe. However, most VLs we deal with in modelling and simulation are graph-like.

Although other tools based on graph grammars (such as DiaGen [21]) use the concept of bootstrapping, in ATOM^3 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 formalism, including graph grammars) and thus the user can change it, obtaining more flexibility.

With respect to graph grammars, an interesting alternative to our approach for formalism transformation is the use of triple graph grammars [22]. In this way, one

could obtain translators from the source to the target formalism and vice versa with the same grammar. The approach is mostly useful for syntax-directed environments, in which the editing actions are specified by means of graph grammar rules. In this way, while the user is building a model in the source formalism, the triple graph grammar creates the equivalent model in the target formalism. That is, 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 LHS must be part of its RHS [22]).

Other kind of model transformation to take into account is that of meta-model evolution. That is, changes made to a meta-model to add new features or constructs to the VL it describes [23]. In that case, one would like a means to translate the models, valid instances of the old meta-model into the syntax of the new one. This can be regarded as a particular case of formalism transformation, and could be modeled with graph transformation rules, as in the example in Section 5.

Other commercial meta-modelling tools, such as DoME [24] or MetaEdit+ [25] 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. As stated before, graph grammars are a high-level, formal and visual notation, which frees the user with the necessity of having to know many details of the internals of the tool, and to maintain textual code.

In the simulation community, there have been very few attempts to combine meta-modelling and graph transformation techniques. For example, in the approach of [26], DoME was used to implement several editors for continuous (sequential function charts) and discrete formalisms (Statecharts). The user builds his composite models with these editors, and they are subsequently translated into the object-oriented simulation language Modelica [27]. This is the only transformation they implement (code generation), and was specified using the textual language Alter. In this area, the approach of *Ptolomey* [28] (which uses the *co-simulation* approach) and *GME* [29] are also worth mentioning. The latter tool has lately incorporated graph grammar techniques for model manipulation [30].

To the authors' knowledge, the systematic approach presented in this paper is novel. The approach combines meta-modelling to define VLs and graph grammars to specify simulators, transformations, optimizations and code generation. This work can thus be considered as an approach for rapid prototyping by means of visual programming.

9. Conclusions

This paper has shown a framework for the generation of visual modelling tools based on meta-modelling and graph grammars. We define the abstract and concrete syntax of the notation using meta-formalisms such as UML class or Entity Relationship Diagrams. With this information, AToM^3 is able to automatically generate a tool to process (load/save/verify its correctness) models in the

described notation. The generated tool's functionality can be further extended by defining manipulations in the graph grammar formalism. This has the advantage of being a high-level, visual and formal notation. As an example of these concepts, we have defined a visual formalism for discrete simulation together with graph grammars for simulation, transformation, optimization and code generation.

In some cases, adding the tool's functionality by means of graph grammars improves productivity as the abstraction level of graph grammars is higher than usual textual programming languages. Additionally, the user does not have to know implementation details of the tool, and the maintenance is done at the level of models, and not of code. In AToM^3, graph grammars are highly reusable, because it is possible to apply the same graph grammar to models in which there is not an exact match with the LHS of the rules, but a subtyping relationship. On the contrary, computations specified by means of graph transformation are usually less efficient than computations specified with lower level programming languages. Nonetheless, in the opinion of the authors, and for the kind of applications to which this approach has been applied, the benefits are bigger than the drawbacks.

AToM^3 has been used by students in pre- and post-graduate courses at McGill and in Madrid for building small projects. Additionally, it has been used to build modelling environments for DEVS [31], OOCSMP [32] and UML (to analyse model properties via model-checking) [6] as well as to create Zope products [31].

Textual languages can still be processed by AToM^3, as one can build a meta-model for the *Abstract Syntax Graph* (the structure used by compilers to store a parsed program). Although possible, manually drawing a graph representing a textual program is not a very appealing approach. While visual languages are naturally described by means of (graphical) meta-models, textual languages are more effectively described using Chomsky string grammars [10]. We are currently working in automating as much as possible the translations of meta-models into string grammars and the opposite operation [33].

Another area that is worth exploring is the applicability of graph grammars in education [11]. Not only are they useful to illustrate Discrete-Event Simulation concepts (such as simulation), but their use is also applicable to teaching in other areas, for example to show the animation of algorithms on data structures such as lists, stacks, vectors, and of course graphs. We are also working in formalising concepts applicable to many formalisms, such as inheritance, hierarchy and multiple abstraction levels. Finally, we want to fully automate the process of multi-formalism modelling.

Acknowledgements

The authors would like to thank the anonymous referees for their accurate and very useful comments. This work has been partially sponsored by the Spanish Ministry of Science and Technology (TIC2002-01948).

References

[1] F.P. Brooks, The Mythical Man Month, Addison-Wesley, Reading, MA, 1995.

[2] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1, World Scientific, Singapore, 1999.

[3] J. de Lara, H. Vangheluwe, AToM^3: a tool for multi-formalism modelling and meta-modelling, in: Proceedings of ETAPS/FASE'02, Lecture Notes in Computer Science, Vol. 2306, Springer, Berlin, 2002, pp. 174–188. See also the AToM^3 home page: http://atom3.cs.mcgill.ca (last visited 2003-03-16).

[4] J.L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[5] G. Gordon, System Simulation, 2nd Edition, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[6] E. Guerra, J. de Lara, A Framework for the Verification of UML Models. Examples using Petri Nets. Jornadas de Ingeniera del Software y Bases de Datos, JISBD, Alicante, Spain, 2003, pp. 325–334.

[7] G.S. Fishman, Discrete event simulation, Modeling, Programming and Analysis, Springer Series in Operations Research, Springer, New York, 2001.

[8] C. Ramchandani, Performance evaluation of asynchronous concurrent systems by timed petri nets, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, 1973.

[9] Minuteman Software Home page: http://www.minutemansoftware.com (last visited 2003-03-16).

[10] A.V. Aho, R. Sethi, J.D. Ullman, Compilers, Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1986.

[11] J. de Lara, Educational simulation by means of meta-modelling and graph grammars, Revista de Enseñanza y Tecnología 23 (2002) 5–17 (in Spanish).

[12] J.B. Warmer, A. Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley Object Technology Services, Reading, MA, 1999.

[13] R. Bardohl, A Visual Environment for Visual Languages, Sci. Comput. Programming 44 (2002) 181–203 See also the GENGED home page: http://tfs.cs.tu-berlin.de/~genged/ (last visited 2003-03-16).

[14] H. Vangheluwe, DEVS as a common denominator for multi-formalism hybrid systems modelling, IEEE Symposium on Computer-Aided Control System Design, Anchorage, UDA, IEEE Computer Society Press, Silver Spring, MD, 2000, pp. 129–134.

[15] P. Fishwick, B.P. Zeigler, A multimodel methodology for qualitative model engineering, ACM Transactions on Modelling and Computing Simulation 1 (2) (1992) 52–81.

[16] R. Razouk, The derivation of performance expressions for communication protocols from timed petri net models, Proceedings of 2nd Conference on Communications Architectures and Protocols, Montreal, Canada, 1984, pp. 210–217.

[17] J.P. Hayes, Computer Architecture and Organization, McGraw-Hill, New York, 1988.

[18] C. Ermel, M. Rudolf, G. Taentzer, The AGG approach: language and tool environment, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rosenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1, World Scientific, Singapore, 1999, pp. 551–604. See also the AGG Home page: http://tfs.cs.tu-berlin.de/agg/ (last visited 2003-03-16).

[19] MDA specification at the OMG's home page: http://www.omg.org (last visited 2003-03-16).

[20] C. Ermel, R. Bardohl, Multiple Views of Visual Behavior Models in GenGed Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 72(3), Elsevier, Amsterdam, 2003.

[21] M. Minas, Bootstrapping visual components of the DiaGen specification tool with DiaGen, Proceedings of AGTIVE'03 (Applications of Graph Transformation with Industrial Relevance), Charlottesville, USA, 2003, pp. 391–406. See also the DiaGen home page: http://www2.informatik.uni-erlangen.de/DiaGen/ (last visited 2003-03-16).

[22] A. Schürr, Specification of Graph Translators with Triple Graph Grammars, in Lecture Notes in Computer Science, Vol. 903, Springer, Berlin, 1994, pp. 151–163.

[23] J. Sprinkle, A. Agrawal, T. Levendovsky, F. Shi, G. Karsai, Domain model evolution in visual languages using graph transformations, 2nd OOPSLA Workshop on Domain Specific Visual Languages, Seattle, USA, 2002.

[24] See the DOME home page: http://www.htc.honeywell.com/dome/ (last visited 2003-03-16), Honeywell Technology Center.
[25] R. Pohjonen, J.-P. Tolvanen, Automated Production of Family Members: Lessons Learned, Proceedings of PLEES'02, Seattle, USA, 2002, pp. 49–57. See also the MetaEdit+ home page. http://www.metacase.com/ (last visited 2003-03-16), MetaCase Consulting.
[26] M. Pereira Remelhe, S. Engel, M. Otter, A. Derarade, P. Mosterman, An environment for integrated modelling of systems with complex continuous and discrete dynamics, Lecture Notes in Control and Information Systems, Vol. 279, 2002, pp. 83–105.
[27] H. Elmqvist, S.E. Mattson, An introduction to the physical modeling language modelica, Proceedings of the 9th European Simulation Symposium ESS'97, SCS International Erlangen, 1997, pp. 110–114. See also http://www.Modelica.org (last visited 2003-03-16).
[28] E.A. Lee, Embedded Software, in: M. Zelkowitz (Ed.), Advances in Computers, Vol. 56, Academic Press, London, 2002. See also: http://ptolomey.eecs.berkeley.edu (last visited 2003-03-16).
[29] A. Lédczi, A. Bakay, M. Marói, P. Vögyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer, November 2001, pp. 44–51. See also the GME home page: http://www.isis.vanderbilt.edu/Projects/gme/default.html (last visited 2003-03-16), Vanderbilt University.
[30] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, On the use of graph transformation in the formal specification of computer-based systems, Workshop on Formal Specification of Computer-Based Systems, Huntsville, USA, 2003, pp. 19–27.
[31] A. Levytsky, E. Kerckhoffs, E. Posse, H. Vangheluwe, Creating DEVS components with the Metamodelling tool AToM^3, Proceedings of ESS (European Simulation Symposium), Society for Modelling and Simulation International, Delft, 2003, pp. 91–103.
[32] J. de Lara, H. Vangheluwe, M. Alfonseca, Meta-modelling and graph grammars for multi-paradigm modelling with AToM^3, Software and Systems Modelling (2003), to appear.
[33] J. de Lara, E. Guerra, Towards the uniform manipulation of visual and textual languages in AToM^3, in: Proceedings of PROLE, Alicante, Spain, 2003, pp. 45–59.