Programmed Graph Rewriting with DEVS

Eugene Syriani and Hans Vangheluwe

School of Computer Science
McGill University, Montréal, Québec, Canada

Abstract. In this article, we propose to use the Discrete EVent system Specification (DEVS) formalism to describe and execute graph transformation control structures. We provide a short review of existing programmed graph rewriting systems, listing the control structures they provide. As DEVS is a timed, highly modular, hierarchical formalism for the description of reactive systems, control structures such as sequence, choice, and iteration are easily modelled. Non-determinism and parallel composition also follow from DEVS' semantics. The proposed approach is illustrated through the modelling of a simple PacMan game, first in AToM^3 and then using DEVS. We show how the use of DEVS allows for modular modification of control structure.

1 Introduction

In 1996, Blostein et al.[1] described some issues regarding the, at that time very sporadic, practical use of graph rewriting. Graphs are a versatile and expressive data representation, and there are many advantages to the explicit representation (as opposed to encoding in the form of programs) of graph transformations. Issues such as expressiveness, scale-ability and re-use of models of graph transformation as well as the ability to integrate such models with traditional software components were considered critical enablers for wide-spread use of graph transformations. During the last decade, several of these issues have been addressed and tools have been developed. In particular, tools such as FUJABA [2] allow for *programmed graph rewriting.* The purpose of programmed graph rewriting is to be able to model the control structure of (graph) transformation. This is done in terms of control flow primitives such as *sequence*, *branching* (choice), and *looping* (iteration). *Hierarchical encapsulation* allows for *modular construction* (and re-use) of control flow structures. Some tools add expressiveness through *non-determinism* and *parallel composition.* In general, it is also desirable for a control structure language to be target (programming) language *neutral.* The explicit incorporation of *time* is rare in current tools. The above requirements were summarized recently in [3].

In our quest for the most appropriate formalism (*i.e.,* which optimally satisfies the above requirements) to describe programmed graph transformation, we now briefly present the features of tools with programmed graph transformation capabilities, based on [4]. Note that our own AToM^3 [5,6], "A Tool for Multi-formalism and Meta-Modelling" which has very limited (priority-based) control structuring, will be introduced in section 3.

Graph Rewriting and Transformation (GReAT). [7,8,9] treats the source model, the target model and the temporary objects created during the transformation as a single graph using a unified metamodel.

The GReAT graph transformation language uses the Single Pushout algebraic approach for subgraph matching. Rules consist of a pattern graph described using UML Class Diagram notation where the elements can be marked to match a pattern (`Bind role`), to remove elements (`Delete role`) or to create elements (`CreateNew role`). A guard is associated with each production; this is an OCL expression that operates on vertex and edge attributes. An attribute mapping can also be defined to generate values of vertex and edge attributes with arithmetic and string expressions.

GReAT's control flow language uses a control flow diagram notation where a production is represented by a block. Sequencing is enabled by the use of input and output interfaces (`Inports` and `Outports`) of a block. Packets (the graph model) are fed to productions via these ports. The `Inport` also provides an optimization in the sense that it specifies an initial binding for the start of the pattern matcher. Two types of hierarchical rules are supported. A `block` pushes all its incoming packets to the first internal rule, whereas a `forblock` pushes one packet through all its internal rules. Branching is achieved using test case rules, consisting of a left-hand side (LHS) and a guard only. If a match is found, the packet will be sent to the output interface. Parallel execution is possible when the `Outports` of a production are connected to different `Inports`. There is no notion of time.

Visual Modelling and Transformation System (VMTS). In VMTS [3,10], the LHS and right-hand side (RHS) of a graph transformation rule are represented as two separate graphs. They can be linked (internal causality) by XSL scripts. These scripts allow attribute operations and represent the `create` and `modify` operation of the transformation step. Also, parameters and pivot nodes can be passed to a step for optimization.

The programmed graph rewriting system of VMTS is the VMTS Control Flow Language (VCFL), a stereotyped Activity Diagram. This abstract statemachine handles pre- and post-conditions of rules. Sequencing is achieved by linking transformation steps; loops are allowed. Branching in VCFL is conditioned by an OCL expression. In case of multiple branching (step connected to more than one step), only the first successfully evaluated branch will apply its transformation step. Iteration is controlled by loops in a sequence of steps. A branch can also be added to provide conditional loops. Hierarchical steps are composed of a sequence of primitive steps. A primitive step ends with success if the terminating state is reached and ends with failure when a match fails. However, in hierarchical steps, when a decision cannot be found at the level of primitive steps, the control flow is sent to the parent state or else the transformation fails. Parallelism is not yet implemented in VCFL. VMTS is language-oriented towards the .NET framework. There is no notion of time.

PROGReS, FUJABA and MOFLON. The PROgrammed Graph REwriting System (PROGReS) [11,12] was the first fully implemented environment to allow programming through graph transformations. It has very advanced features not found in other tools such as back-tracking. Insights gained through the development of PROGReS have led to FUJABA (From UML to Java and Back Again) [2,13], a completely redesigned graph transformation environment based on Java and UML. FUJABA's programmed graph rewriting system is based on Story Charts, a combination of Story Diagrams [13] and Statecharts. An activity in such a diagram contains either graph rewrite rules, which adopt Collaboration Diagram-like representation, or pure Java code. The graph schemes for graph rewriting rules exploit UML class diagrams. With the expressiveness of Story Charts, graph transformation rules can be sequenced (using success and failure guards on the linking edges) along with activities containing code. Branching is ensured by the condition blocks which act like an if-else construct. An activity can be a for-all story pattern, which acts like a while loop on a transformation rule.

FUJABA's approach is implementation-oriented. Classes define method signatures and method content is described by Story Chart diagrams. All models are compiled to Java code. There is no notion of time.

The MOFLON [14] toolset uses the FUJABA engine for graph transformation, since the latter already features UML-like graph schemata. It provides an environment where transformations are defined by Triple Graph Grammars (TGGs) [15]. These TGGs are subsequently compiled to Story Diagrams [13]. This adds declarative power to FUJABA similar to that of the OMG's QVT (Query/View/Transformation – `www.omg.org`).

In the sequel, we propose the Discrete EVent system Specification (DEVS) formalism [16] to describe transformation control structures. Using DEVS gives us sufficient expressiveness to match that of the tools described above, thus satisfying the requirements for transformation control structure description languages listed before. Furthermore, as with the adaptation of known formalisms such as Activity Diagrams in tools such as FUJABA, using DEVS means that no new formalism needs to be invented (and its properties investigated). Also, existing tools for analysis, simulation, and code synthesis may thus be re-used for the control structure part of a graph transformation model.

The remainder of this paper is structured as follows. Section 2 describes the DEVS formalism. Section 3 describes PacMan, a small case study, and how it is modelled in AToM^3. Section 4 describes how the priority-based graph rewriting semantics of AToM^3 can be modelled using DEVS. The combination of DEVS with Graph Rewriting rules is very elegant and orthogonal. It is shown how the modularity of DEVS allows for easy modification of the transformation control structure. This modification includes the specification of real-time user interaction. Section 5 describes the advantages of using DEVS for programmed graph transformation and section 6 summarizes and concludes.

2 Discrete Event System Specification (DEVS)

This section introduces the *Discrete EVent system Specification* (DEVS) formalism. In the rest of the paper, it will be shown how the modularity and expressiveness of DEVS are well suited to encapsulate graph rewriting building blocks.

The DEVS formalism was introduced in the late seventies by Bernard Zeigler to develop a rigorous basis for the compositional modelling and simulation of discrete event systems [16]. The DEVS formalism has been successfully applied to the design, performance analysis and implementation of a plethora of complex systems.

A DEVS model is either *atomic* or *coupled*. An atomic model describes the behaviour of a reactive system. A coupled model is the composition of several DEVS sub-models which can be either atomic or coupled. Submodels have *ports*, which are connected by channels. Ports are either *input* or *output*. Ports and channels allow a model to receive and send signals (events) from and to other models. A channel must go from an output port of some model to an input port of a different model, from an input port of a coupled model to an input port of one of its sub-models, or from an output port of a sub-model to an output port of its parent model.

An **atomic DEVS**[1] model is a tuple $(S, X, Y, \delta^{int}, \delta^{ext}, \lambda, \tau)$ where S is a set of sequential **states**, one of which is the *initial* state. X is a set of allowed **input events**. Y is a set of allowed **output events**. There are two types of transitions between states: $\delta^{int} : S \rightarrow S$ is the **internal transition function**, $\delta^{ext} : Q \times X \rightarrow S$ is the **external transition function**, Associated with each state are $\tau : S \rightarrow \mathbb{R}_0^+$, the **time-advance** function and $\lambda : S \rightarrow Y$, the **output function**. In this definition, $Q = \{(s, e) \in S \times \mathbb{R}^+ \mid 0 \leq e \leq \tau(s)\}$ is called the **total state space**. For each $(s, e) \in Q$, e is called the **elapsed time**. $\mathbb{R}_0^+$ denotes the positive reals with zero included.

Informally, the operational semantics of an atomic model is as follows: the model starts in its initial state. It will remain in any given state for as long as the time-advance of that state specifies or until input is received on some port. If no input is received, after the time-advance of the state expires, the model first (before changing state) sends output as specified by λ, and then instantaneously jumps to a new state specified by δ^{int}. If input is received however before the time for the next internal transition, then it is δ^{ext} which is applied. The external transition depends on the current state, the time elapsed since the last transition and the inputs from the input ports.

The following definition formalizes the concept of coupled DEVS models. A **coupled DEVS**[1] model named D is a tuple $(X, Y, N, M, I, Z, select)$ where X is a set of allowed **input events** and Y is a set of allowed **output events**. N is a set of **component names** (or labels) such that $D \notin N$. $M = \{M_n \mid n \in N, M_n$ is a DEVS model (atomic or coupled) with input set X_n and output set $Y_n\}$ is a set of DEVS **sub-models**. $I = \{I_n \mid n \in N, I_n \subseteq N \cup \{D\}\}$ is a set of **influencer** sets for each component named n. I encodes the connection topology of sub-models. $Z = \{Z_{i,n} \mid \forall n \in N, i \in I_n . Z_{i,n} : Y_i \rightarrow X_n \textit{ or } Z_{D,n} : X \rightarrow X_n \textit{ or } Z_{i,D} : Y_i \rightarrow Y\}$ is a set of **transfer functions** from each component i to some component n. $select : 2^N \rightarrow N$ is the **select** or tie-breaking function. 2^N denotes the powerset of N (the set of all sub-sets of N).

[1] For simplicity, we do not present a formalization of the concept of "ports".

The connection topology of sub-models is expressed by the influencer set of each component. Note that for a given model n, this set includes not only the external models that provide inputs to n, but also its own internal sub-models that produce its output (if n is a coupled model.) Transfer functions represent output-to-input translations between components, and can be thought of as channels that make the appropriate type translations. For example, a "departure" event output of one sub-model is translated to an"arrival" event on a connected sub-model's input. The *select* function takes care of conflicts as explained below.

The semantics for a coupled model is, informally, the parallel composition of all the sub-models. A priori, each sub-model in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur except if it is explicitly modelled by the output function of a sender, and the external transition function of a receiver. There is however a *serialization* whenever there are multiple sub-models that have an internal transition scheduled to be performed at the same time. The modeller controls which of the conflicting sub-models undergoes its transition first by means of the *select* function.

We have developed our own DEVS simulator called `pythonDEVS` [17], grafted onto the object-oriented scripting language Python. In a recent M.Sc. thesis [18], a compiler for Modelica (`www.modelica.org`) textual representations of DEVS models as well as a visual modelling environment were developed.

3 A Small Case Study: PacMan in AToM^3

In this section, we describe the simple priority-based graph rewriting in our meta-modelling and model transformation tool AToM^3 [5,6]. In the next section, this hard-coded control structure will be modelled explicitly using DEVS. As an example, we use a simplified version of the PacMan video game used in Heckel's tutorial introduction of graph transformation [19].

3.1 The PacMan Language (Abstract and Concrete Syntax)

The PacMan language has five distinct elements: PacMan, Ghost, Food, GridNode and ScoreBoard. Fig. 1 shows the meta-model (model of the abstract syntax) of this modelling language in AToM^3. PacMan, Ghost and Food objects can be linked to GridNode objects; note the use of associations. This depicts that these objects can be "on" a gridNode. The self-association between GridNode objects represents the geometric organization of the game area, similar to the classical PacMan video game. At a semantic level, this will also denote that PacMan and Ghost "may move" to a connected gridNode. A Scoreboard object holds an integer valued attribute score. The reason for having different associations from the classes to the GridNode class is for concrete visual syntax purposes. AToM^3 allows one to associate a visual representation to each class and association. Associations can be concretely represented visually by means of arrows or by a geometric/topological constraint relation, such as a PacMan being centered over a GridNode. Note how in this example there are no restrictions on the number of instances of each element, nor on the number of links to a GridNode instance.

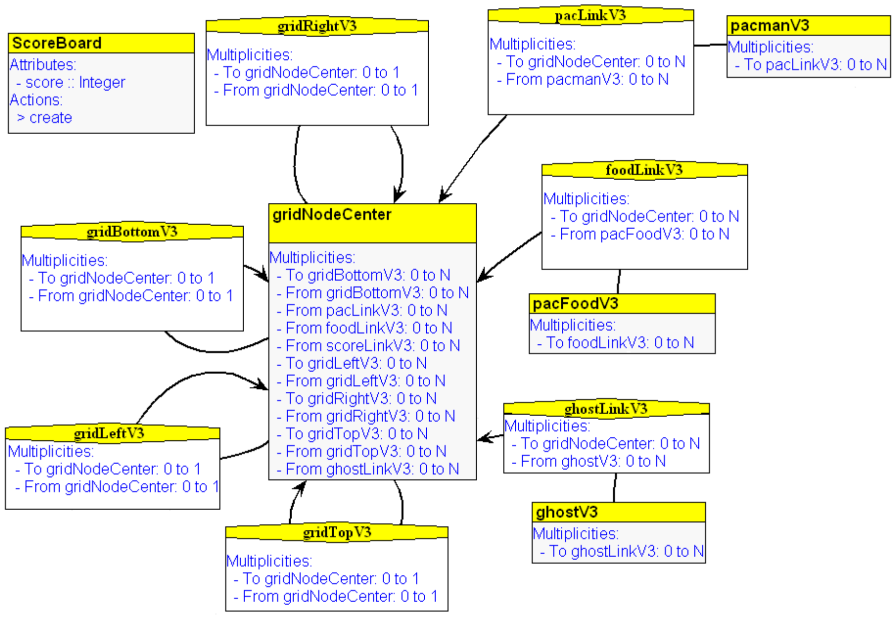

Fig. 1. The PacMan Meta-Model

3.2 The PacMan Semantics (Graph Grammar)

The operational semantics of the PacMan formalism is defined in a Graph Grammar model which consists of a number of rules. In the rules in the following figures, concrete syntax is used. this is a useful feature for domain-specific modelling unique to AToM^3. Dashed lines were added to explicitely show the "on" links. Rule 1 in Fig. 2 shows killing: when a Ghost object is on a GridNode which has a PacMan object, the PacMan is removed. Rule 2 in Fig. 3 shows eating: when a PacMan object is on a GridNode which has a Food object, Food is removed and the score gets updated (using an attribute update expression). Rule 3 in Fig. 4 expresses the movement of a Ghost object to the right and rule

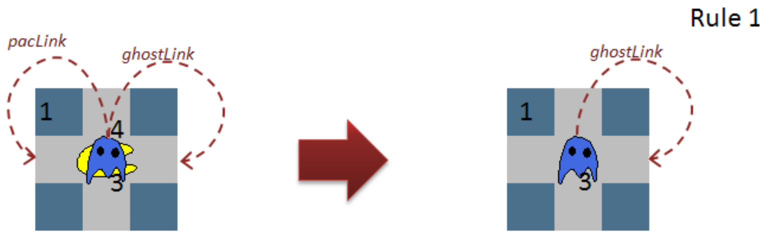

Fig. 2. PacMan Semantics: Ghost kills PacMan rule

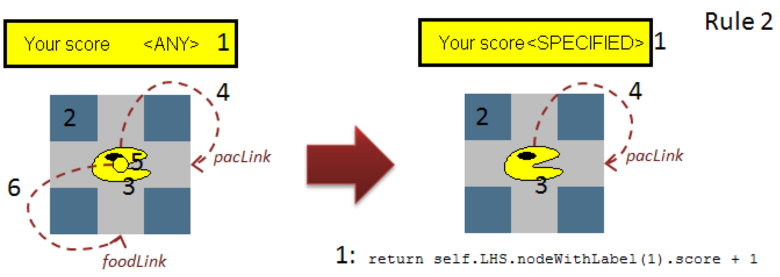

Fig. 3. PacMan Semantics: PacMan eats Food rule

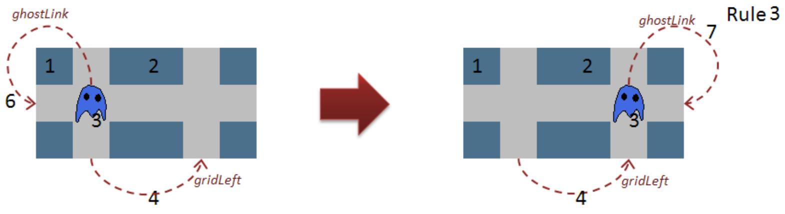

Fig. 4. PacMan Semantics: Ghost moves right rule

8 in Fig. 5 the movement of a PacMan object to the left. Similar rules to move Ghosts and PacMan objects up, down, left and right are part of the grammar but are not shown. Rules 1 and 2 have priorities 1 and 2 respectively. All remaining rules have the same priority 3.

3.3 AToM^3's Graph Grammar Semantics

AToM^3's graph rewriting engine supports priority-based execution of rewrite rules. Rules are grouped based on their priority. Rewriting starts with the highest-priority group of rules. If for at least one of the rules in the group, a match if found in the host graph, one of those rules is chosen non-deterministically. Subsequently, the re-write is performed on the host graph and control goes back to the group of rules with the highest priority. If none of the rules in the group

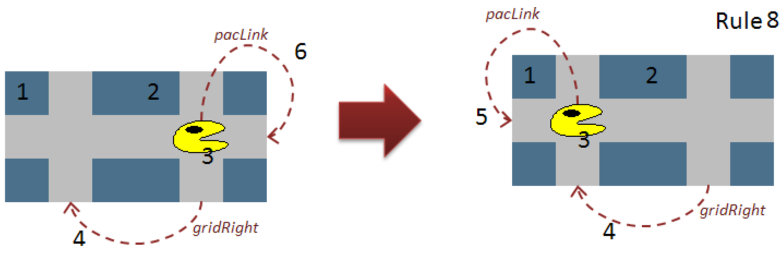

Fig. 5. PacMan Semantics: PacMan moves left rule

match, control goes to the group of rules with the next lower priority, and so on. If no groups of rules are left (even the lowest-priority rules do not yield a match), the transformation terminates. In AToM^3, execution can be done step-by-step (for simulation purposes) or in continuous mode (useful for terminating model-to-model transformations). Note that AToM^3 allows the specification of (real-)time taken by each rule-rewrite. The time may be extracted from model attributes. This allows for meaningful simulation animation.

4 Programmed Graph Rewriting Using DEVS

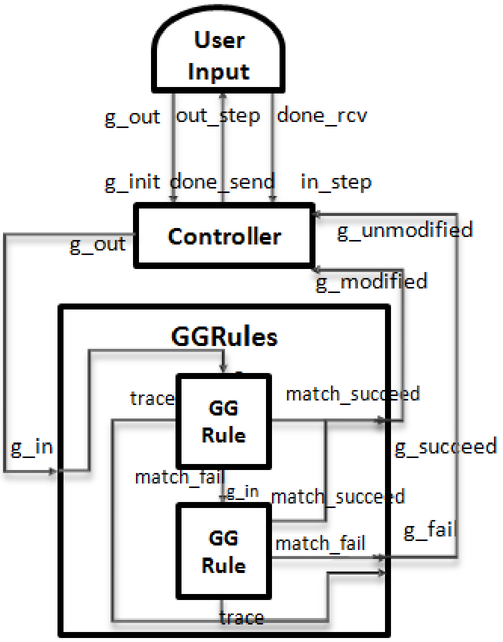

Fig. 6. The overall coupled DEVS model

The purpose of programmed graph rewriting is to explicitly model the control flow of (graph) transformation. This is done in terms of control flow primitives such as sequence, choice, and looping. Hierarchical encapsulation allows modular construction (and re-use) of control flow structures. Some tools increase expressiveness through constructs such as non-determinism and parallelism. Rather than inventing a new language for control structure description, we propose to use the DEVS formalism, with its precisely defined syntax and semantics, presented above.

As an illustration of how this approach satisfies the requirements stated earlier, we explicitly model AToM^3's Graph Transformation execution engine described above. The starting point of our approach is to encapsulate to-be-transformed graphs in DEVS events. These events will be sent between the DEVS building blocks encoding graph transformation rules. Additionally, events may encode control signals which can be sent to designated ports of DEVS building blocks. Only atomic DEVS models perform actual transformations. Coupled DEVS models allow one to hierarchically construct complex transformation models. Atomic DEVS models are highly encapsulated (they can only communicate via their input- and output-ports) and can be used to represent a variety of models in different formalisms, ranging from code (in the target language of the DEVS simulator used – Python in our case) to Statecharts. The only constraint is that building blocks need to accept graphs on their input port and, after transformation, produce graphs on their output port. The topology of the coupled DEVS models encodes the control structure. As a result of the DEVS semantics, the flow of events (graphs) through a DEVS coupled model resembles data flow more than control flow. In the construction which follows, we will only allow one model to flow through the network at any time. This effectively makes data flow and control flow identical. In the future, we will however exploit the data flow nature of DEVS networks, in particular for parallel implementations.

Fig. 6 shows the overall structure of the DEVS model for AToM^3-style graph transformation. Each block is shown with its ports along with the connections. Execution (transformation) is triggered by some user control. User intervention (such as a possible interruption of a running simulation) is modelled in the UserInput block. Note that the DEVS formalism allows one to specify external pre-emptive interrupts through the external transition function. The Controller block acts as the interface of the transformation system to the user: it receives user inputs and informs the user of the status of the execution. It also models the transformation steps management. The GGRules block receives the host graph from the Controller and returns the transformed graph. The Python code below (synthesized from the control flow model given in Fig. 6) shows a small part of the `pythonDEVS` representation of the overall model. Instances of atomic DEVS building blocks corresponding to the control flow model building blocks are `connect`ed. Note that in our implementation, we have added a Trace atomic DEVS block to log all transformation steps.

```python
class PacManGGExec(CoupledDEVS):
def __init__(self, graph, steps):
  self.USERINPUT = self.addSubModel(UserInput(graph=graph, steps=steps))
  self.CONTROLLER = self.addSubModel(Controller())
  self.RULES = self.addSubModel(GGRules())
  self.TRACE = self.addSubModel(Trace())
  self.connectPorts(self.USERINPUT.g_out, self.CONTROLLER.g_init)
  self.connectPorts(self.USERINPUT.out_step, self.CONTROLLER.in_step)
  self.connectPorts(self.CONTROLLER.done_send, self.USERINPUT.done_rcv)
  self.connectPorts(self.CONTROLLER.g_out, self.RULES.g_in)
  self.connectPorts(self.RULES.trace, self.TRACE.in_rule)
```

4.1 The User Input Block

UserInput is an atomic DEVS block that sends graphs and “steps” control signals and receives termination events. The graphs are Abstract Syntax Graphs (ASGs), AToM^3's basic internal data structure, of models in the PacMan language. Steps represent the number of steps the user requests the simulator to perform in a row. 0 ends the simulation. ∞ runs the simulation in continuous mode, executing till termination (or until interrupted by an external signal). The reception of a Termination event means that either the requested number of steps have been performed or that the execution has reached its end. In the latter case, no more transformations can be applied to the graph. The inports and outports of the UserInput block are connected to the Controller block.

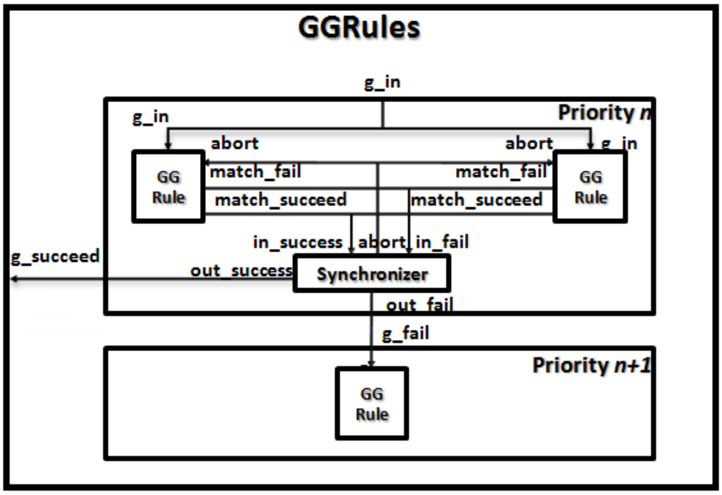

Fig. 7. Managing priorities

4.2 The Controller Block

The Controller atomic DEVS block encodes the coordination logic between the external input and the transformation model. It is the control that receives the graph to transform and the number of steps to be applied. It also notifies the user about termination. The Controller sends the graph to the transformation model and waits for a graph in return. The returned graph may or may not be modified. This is repeated depending on the "steps" requests received. Note that the system could in principle receive multiple graphs at any time (thanks to the data flow nature of DEVS). Also, the user could request more "steps" even when there are some steps left in the running transformation.

4.3 The Graph Grammar Rule Blocks and Priority

The graph rewriting rules presented in section 3.2 including the semantics of prioritized rewriting, are encoded in the transformation block GGRules. GGRules is a coupled DEVS model which receives a graph and outputs a graph. GGRules is composed of one or more GGRule blocks. Each GGRule satisfies certain properties. There is at most one rule that is applied per step. If a rule fails, the graph is sent to the next rule until the last rule is reached. If the last rule also fails, then no rules have been applied in this step, hence GGRules sends back its input graph. Otherwise it is the newly transformed graph that is sent back, directly from the rule where the match occurred.

The AToM^3 graph rewriting system allows assigning priorities to rules to order their execution. If multiple rules happen to have the same priority, AToM^3 non-deterministically chooses one of those yielding a match. A Synchronizer block is introduced to model this situation in our DEVS model. This is depiced in Fig. 7.

All rules with the same priority (also known as a layer) will receive their input in parallel from the previous layer. A failed matching of a rule is notified to the Synchronizer. If it has received failure notices from all rules in the layer, it passes the input to the next layer. On the other hand, as soon as one rule has successfully executed, it notifies the Synchronizer which, in turn, aborts the execution of the remaining rules. It then sends the output to GGRules. As long as the content of a GGRule block is a valid atomic DEVS and it accepts and returns ASGs, it can be arbitrary (hand-coded, compiled or interpreted from some specification). In the case of this example, we compile each AToM^3 PacMan rule into an `execute` method used inside an atomic DEVS external transition. A small excerpt of the code synthesized from the rule to match the LHS of the Kill rule is given below:

```
class Kill(Rule):
  def execute(self, graph):
    # Find matching subgraph #
    match = 0
    try:
      for ghost in graph.listNodes['GhostV3']:
        for ghostLink in ghost.out_connections_:
          if ghostLink.__class__.__name__ == 'GhostLinkV3':
            for pacman in contains.out_connections_:
              if pacman.__class__.__name__ == 'PacmanV3':
                for pacLink in contains.out_connections_:
                  if pacLink.__class__.__name__ == 'PacLinkV3':
                    match = 1 # First occurence of the subgraph
                    break
              if match: break
          if match: break
        if match: break
    except:
      return None
    if not match:
      return None
    # Transform subgraph #
     ....
    return graph
```

4.4 Extending the Model

To illustrate the power of this formalism to describe control flow of graph rewrinting systems, we now *extend* the previous model. Consider the PacMan formalism described in section 3 and the graph grammar that described its behaviour. Suppose we would like more interaction with the user. In the model used before, the simulation could be triggered by the user specifying the numbers of steps to be performed or continuous execution (till termination). We will now allow user control of PacMan movement to more closely mimic the behaviour of the classic PacMan video game. Fig. 8 shows the extended model. The UserInput block remains unchanged, with an outport added. The user can now send a pressed Key code to the Controller block. This enables us to simulate the user interrupts to move the PacMan up, down, left or right. The behaviour of the Controller block is the same as long as no Key is recieved. If this event occurs however, the Controller waits for the reception of a graph from the transformation block(s) and then sends the Key and the Graph to the UserControlledRules block. Otherwise, graphs are always sent to the AutonomousRules block. The AutonomousRules encapsulates all the rules that do not need user intervention: PacMan eating, Ghost killing PacMan and Ghost moving. The structure of this block is exactly the same as the original GGRules block.

The UserControlledRules model consists of the remaining rules, those resposible for PacMan movement (left, right, up, down). This coupled DEVS block recieves a Key and an input graph and outputs a graph that has undergone the requested transformation. Fig. 9 presents the content of this block. The received Key goes through a Dispatch block. This block choses where to send the graph depending on the key pressed. The graph is sent to at most one of the Up, Down, Left and Right blocks. These blocks have the exact same structure as their counterpart in the original GGRule model. Note that event-based selection of rules has previously been called "Event-driven Graph Rewriting".

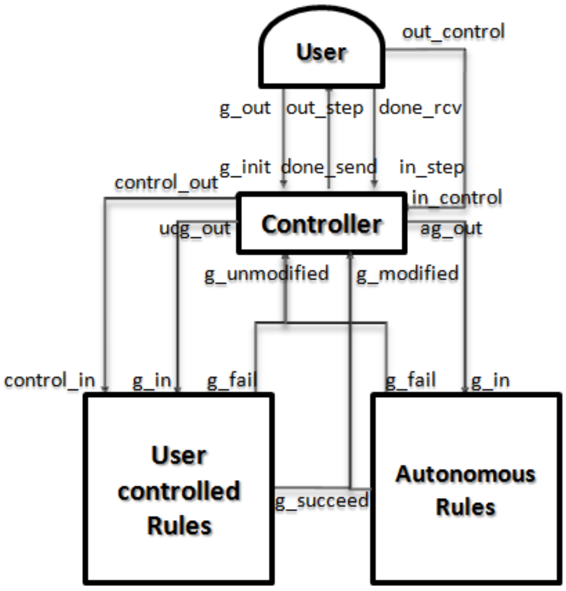

Fig. 8. The Extended DEVS model

With the first model, we showed how to model a simulator for graph grammar execution to mimic the AToM^3 behaviour. Then, we showed how to extend continuous execution with user control over the execution. Note how the extension of the former model needed very little effort thanks to the modularity of DEVS blocks and the ability of DEVS to represent interrupts. Only adding blocks and connections but no modification of any original blocks was needed.

5 Advantages of Using DEVS

The approach described above elegantly satisfies all the requirements enumerated at the beginning of this paper.

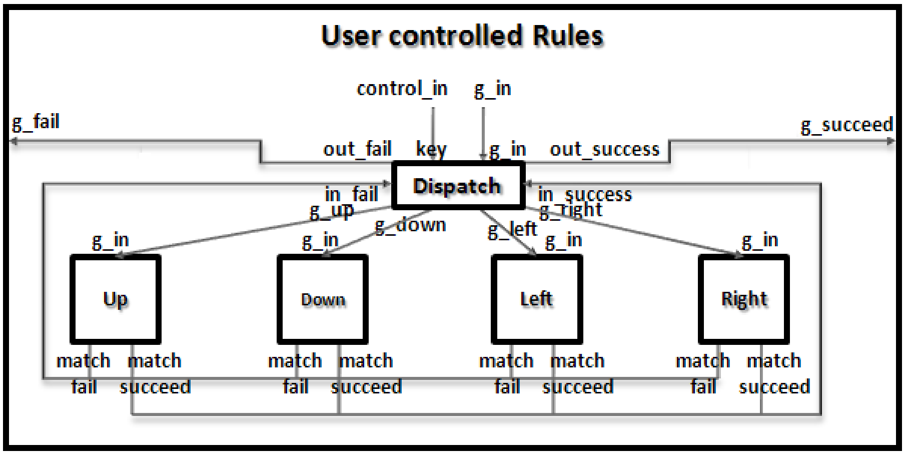

Fig. 9. The User-controlled Rules block

The Power of DEVS. The transformation language used in the PacMan example emulates AToM^3's rewriting semantics. In fact, we could have used another graph transformation semantics (such as unordered or layered graph rewriting). Note that the approach has the potential to support features such as backtracking as in PROGReS. We could even have combined different transformation specification languages. As such, DEVS acts as a "glue" language.

The power of DEVS lies in the ability to express the control flow of the transformation. Each rule is represented in an atomic-DEVS block (this is comparable to the atomicity of the rules in PROGReS). Blocks receive graphs and sends graph through their ports. Other ports can be added to for example send optimization hints (such as pivot nodes in GReAT and VMTS) or to pass some information on the flow of the rule set (like the Key in the extended PacMan model). DEVS allows modularity. Indeed, coupled DEVS blocks can be treated as black boxes. The use of DEVS allows for multi-level hierarchies in models. Sequencing is treated as in GReAT by simply connecting block ports. Iteration and loops can thus be modelled. A given block can be a test block for branching if we give it such a semantics (*i.e.,* no transformation occurs). This is what the Dispatch block in the PacMan example depicts. Parallel execution is provided by the DEVS formalism when an output port is connected to many input ports. If execution (not simulated) parallelism is needed, the parallel DEVS [20] formalism can be used.

Using the DEVS formalism as a control flow language for graph rewriting enabled us to not only model the AToM^3 simulator for graph grammar execution but also to provide an improved version of it which combined continuous execution and user interaction. Note that we are thus modelling control structures supporting step by step simulation, continuous simulation and user controlled simulation which are not in the system under study, but rather in the execution environment.

Scalability and Multi-Formalism Modelling. The beauty of DEVS models lies in the modularity of its building blocks. In fact, each block performs an action given some input and can produce outputs. This modularity trivially supports the combination of building blocks specified using *multiple formalisms*. Hence, we may combine graph grammars with for example Statecharts and code. This is the key to scaling up (graph) transformation modelling to arbitrarily more complex models, far beyond the limits of pure graph grammar systems.

Modelling Time. Timed Graph Transformation, as proposed by Gyapay, Heckel and Varró [21] integrates time in the double push-out approach. They extend the definition of a production by introducing, in the model and rules, a chronos element that stores the notion of time. Rules can monotonically increase the time. DEVS is inherently a timed formalism, as explained in section 2. In contrast with [21], it is the execution of a rule that can increase time and not the rule itself. Hence, the control flow (of the graph transformation) has full access to it. As pointed out in [21], time can be used as a metric to express how many time units are consumed to execute a rule. Having time at the level of the block containing a rule rather that in the rule itself does not lose this expressiveness. Also, providing time to the control flow structure can enhance the semantics of the transformation. AToM^3 for example provides control over execution time delay for animation (see section 3). In the PacMan example, when modelling the user we can give meaning to the time delay between the execution of different rules. As an example, the autonomous rules may take more time than the user controlled rules moving PacMan. This gives more time for the user to “interrupt”. But if, for instance, the ghost-moving rules take less time, then the user needs to interrupt faster to move PacMan. This becomes closer to a game especially if a real-time simulator such as RT-DEVS [22] is used.

6 Conclusions and Future Work

In this article, we have introduced the Discrete Event system Specification (DEVS) formalism to describe and execute graph transformation control structures. We provided a short review of existing programmed graph rewriting systems, listing the control structures they provide. As DEVS is a timed, highly modular, hierarchical formalism for the description of reactive systems, control structures such as sequence, choice, and iteration are easily modelled. Non-determinism and parallel composition also follow from DEVS’ semantics. The proposed approach was illustrated through the modelling of a simple PacMan game, first in AToM^3 and then with DEVS. We showed how the use of DEVS ultimately allows real-time simulation/execution.

We plan to further investigate the use of DEVS. This will include various types of code synthesis from rules on the one hand and visual control structure specifications on the other hand, beyond our current non-optimized prototype. We also consider mapping our control flow formalism onto formalisms other than DEVS, more suited for real (as opposed to simulated) parallel execution.

We plan to completely model our AToM^3 environment in DEVS. We will then be able to explicitly model users interacting with a transformation environment. This will allow for automated testing of interactive transformations as well as for optimization of transformation models for different types of users.

As consistency is a very important issue in modelling, we plan to integrate Triple Graph Grammars [15] in our DEVS framework. This will allow model synchronization and bi-directional transformations.

Acknowledgments

The authors thank the participants of the 2007 Bellairs CaMPAM workshop for stimulating discussions on programmed graph rewriting as well as the AGTIVE conference attendees for insightful comments. The Natural Sciences and Engineering Research Council (NSERC) of Canada is gratefully acknowledged for partial support of this work.

References

1. Blostein, D., Fahmy, H., Grbavec, A.: Issues in the practical use of graph rewriting. In: Selected papers from the 5th International Workshop on Graph Grammars and Their Application to Computer Science, pp. 38–55. Springer, Heidelberg (1996)
2. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: ICSE 2000: Proceedings of the 22nd International Conference on Software Engineering, pp. 742–745. ACM Press, New York (2000)
3. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Control flow support in metamodel-based model transformation frameworks. In: EUROCON 2005 International Conference on "Computer as a tool", pp. 595–598. IEEE, Los Alamitos (2005)
4. Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varró, D., Varró-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005), Montego Bay, Jamaica (2005)
5. de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) ETAPS 2002 and FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
6. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. Software and Systems Modeling (SoSyM) 3, 194–209 (2004)
7. Vizhanyo, A., Agrawal, A., Shi, F.: Towards generation of high-performance transformations. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 298–316. Springer, Heidelberg (2004)
8. Agrawal, A.: Metamodel based model transformation language. In: OOPSLA 2003: Companion of the 18th annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pp. 386–387. ACM Press, New York (2003)

9. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. Software and Systems Modeling (SoSyM) 5, 261–288 (2005)
10. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model transformation with a visual control flow language. International Journal of Computer Science (IJCS) 1, 45–53 (2006)
11. Blostein, D., Schürr, A.: Computing with graphs and graph rewriting. Proceedings in Informatics, 1–21 (1999)
12. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with progres. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 219–234. Springer, Heidelberg (1995)
13. Fischer, T., Niere, J., Turunski, L., Zündorf, A.: Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
14. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: Moflon: A standardcompliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
15. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
16. Zeigler, B.P.: Multifacetted Modelling and Discrete Event Simulation. Academic Press, London (1984)
17. Bolduc, J.S., Vangheluwe, H.: The modelling and simulation package PythonDEVS for classical hierarchical DEVS. MSDL Technical report MSDL-TR-2001-01, McGill University (2001)
18. Song, H.: Infrastructure for DEVS modelling and experimentation. MSc dissertation, McGill University (2006)
19. Heckel, R.: Graph transformation in a nutshell. In: Proceedings of the School on Foundations of Visual Modelling Techniques (FoVMT 2004) of the SegraVis Research Training Network. Electronic Notes in Theoretical Computer Science (ENTCS), vol. 148, pp. 187–198. Elsevier, Amsterdam (2006)
20. Chow, A.C.H.: Parallel devs: a parallel, hierarchical, modular modeling formalism and its distributed simulator. Transactions of the Society for Computer Simulation International 13, 55–67 (1996)
21. Gyapay, S., Heckel, R., Varró, D.: Graph transformation with time: Causality and logical clocks. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 120–134. Springer, Heidelberg (2002)
22. Hong, J.S., Song, H.S., Kim, T.G., Park, K.H.: A real-time discrete event system specification formalism for seamless real-time software development. Discrete Event Dynamic Systems 7, 355–375 (1997)