DEVS as a Semantic Domain for Programmed Graph Transformation

Eugene Syriani and Hans Vangheluwe

McGill University, School of Computer Science, Montréal, Canada,
{esyria,hv}@cs.mcgill.ca

Abstract. The Discrete Event system Specification (DEVS) formalism allows for highly modular, hierarchical modelling of timed, reactive systems. We formalize graph transformation control structures by expressing them in terms of DEVS models. We also show how this use of DEVS as a semantic domain for controlled rule-based graph transformation allows for simulation and ultimately synthesis of (real-time) applications. Our approach is illustrated by simulating the AntWorld graph transformation benchmark.

1 Introduction

1.1 Meta-Modelling and Model Transformation

Model-driven approaches are becoming increasingly important in the area of software engineering. In model-driven development, models are constructed to conform to meta-models. A meta-model defines the (possibly infinite) set of all well-formed model instances. As such, a meta-model specifies the syntax and static semantics of models. Meta-models are often described as the Unified Modelling Language (UML)'s Class Diagrams. In model-driven engineering, meta-modelling goes hand in hand with model transformation.

In almost all modelling and simulation endeavours, some form of model transformation is used. Models are for example transformed for optimization purposes, to address new requirements, to synthesize real-time embedded code, etc. Transformations are also commonly used to describe the semantics of a modelling formalism. In the case of operational semantics, the transformation iteratively updates the state of a model. In the case of denotational semantics, the transformation maps a model in one formalism onto a model in a known formalism, thereby defining the meaning of the original model.

Model transformations can be described in many ways. Rule-based descriptions are elegant and easy to understand. Such descriptions have declarative (specifying "what" to change, not "how" to) model rewriting rules as their primitive building blocks. A rule consists of a Left Hand Side (LHS) pattern that is matched against a host model. If a match is found, this pattern is updated, in the host model, based on what is specified in the Right Hand Side (RHS) of the rule. Additionally, Negative Application Condition (NAC) patterns may be used, specifying which patterns should not be found in the host model. Because at some level of abstraction, all models can be represented as (typed, attributed) graphs, and thanks to its rigorous formal underpinning, our rule-based specification is based on the theory of graph rewriting.

Though elegant, the declarative, rule-based specifications of model transformations do not scale well. When the number of rules grows, it becomes difficult for a modeller to clearly understand what the behaviour of the transformation will be. Also, the complexity of matching grows with the number of rules that need to be tried. Programmed (or controlled) graph rewriting mitigates these problems. It combines rewriting rules with a control structure. In this chapter we show how the *Discrete EVent system Specification* (DEVS) can be used as a semantic domain for the control structures in a model/graph transformation system.

In Section 2 we introduce our running example, an extended version of a recent benchmark for graph transformation [1]. Section 3 shows how extending the meta-model of DEVS allows for the introduction of programmed (or controlled) model/graph transformation. Then, Section 4 illustrates a solution to the case study problem using that transformation language. Using the notion of time inherent in DEVS, we show how the notion of time can elegantly be added to a transformation ultimately allowing real-time deployment in Section 5. Section 6 compares our DEVS-based approach to other graph transformation approaches. Finally, Section 7 highlights some of the advantages of this approach, summarizes, and concludes. We start with a brief overview of the DEVS formalism.

1.2 The Discrete Event System Specification

This section introduces the DEVS formalism. In the rest of the paper, it will be shown how the modularity and expressiveness of DEVS allow for elegant encapsulation of model transformation (*i.e.*, 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 [2]. It has been successfully applied to the design, performance analysis, and implementation of a plethora of complex systems.

Figure 1 shows the meta-model of a model transformation language based on DEVS. The dashed-line elements in Fig. 1 (for now ignore the full-lined elements) show a simplified meta-model of DEVS in UML Class Diagram notation. A DEVS model (the abstract class *Block*) is either an *AtomicBlock* or a *CoupledBlock*. An atomic model describes the behaviour of a timed, reactive system. A coupled model is the composition of several DEVS sub-models that can be either atomic or coupled. Sub-models have *ports* that are connected by channels (represented by the associations between the different ports). Ports are either *Inport* or *Outport*. The abstract classes *(In/Out)port* can be instantiated in an *Atomic(In/Out)port* or a *Coupled(In/Out)port*, respectively. Ports and channels allow a model to receive and send events (any subclass of *Event*) 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, as depicted by the associations of Fig. 1. Note that the dynamic semantics of DEVS cannot be expressed by the meta-model and will be informally outlined hereafter.

An *atomic DEVS* model is a structure $\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$ 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* and $\delta^{ext} : Q \times X \rightarrow S$ 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, $Q = \{(s, e) \,|s \in S, 0 \leq e \leq \tau(s)\}$ 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 an input 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 the *outputFunction*, and then instantaneously jumps to a new state specified by the *internalTransition*. If, however, input is received before the time for the next internal transition, then the *externalTransition* is applied. The external transition depends on the current state, the time elapsed since the last transition, and the inputs from the input ports.

A *coupled DEVS*[1] model named D is a structure $\langle X, Y, N, M, I, Z, select \rangle$ 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.

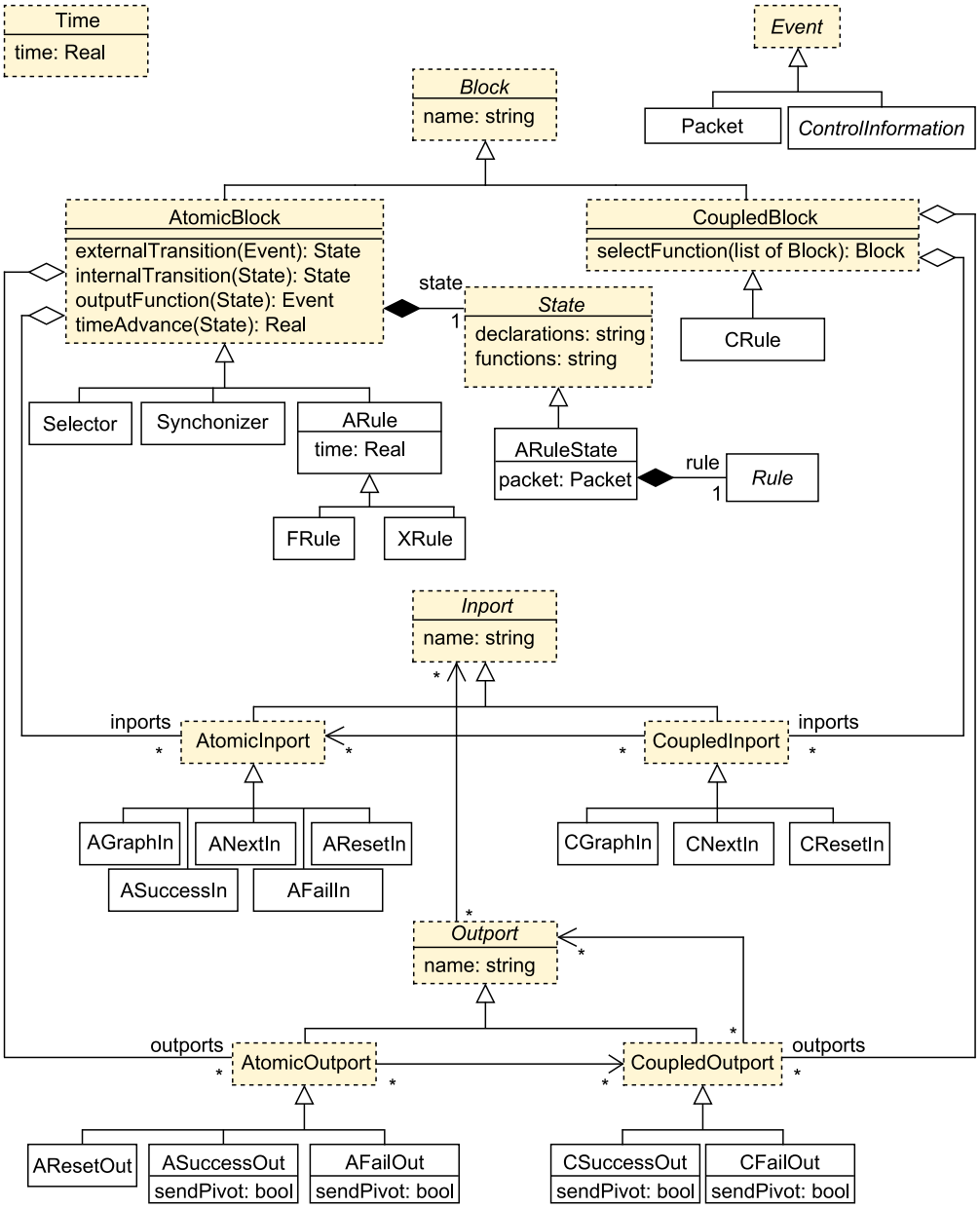


Fig. 1. The MoTif meta-model, based on the DEVS meta-model

$Z = \{Z_{i,n} \mid \forall n \in N, i \in I_n . Z_{i,n} : Y_i \rightarrow X_n \text{ or } Z_{D,n} : X \rightarrow X_n \text{ or } Z_{i,D} : Y_i \rightarrow Y\}$ 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).
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 into 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` [3], grafted onto the object-oriented scripting language Python.

2 The AntWorld Simulation Case Study

The case study used in this chapter is based on case no. 2 (AntWorld Simulation case study) of the GraBaTs 2008 tool contest [1]. This is a benchmark for the comparison of graph transformation tools that stresses local rule application. A solution using a DEVS-based graph transformation language MoTif (Modular Timed graph transformation) [4] was presented at the 2008 tool contest.
The complete description of the behaviour can be found in [1] and is as follows. The AntWorld simulation map is discretized into concentric circles of nodes (representing a large area) centered at a hill (the ant home). Ants are moving around searching for food. When an ant finds food, it brings it back to the ant hill in order to grow new ants. On its way home, the ant drops pheromones marking the path to the food reservoir. If an ant without food leaves the hill or if a searching ant hits a pheromone mark, it follows the pheromone path leading to the food. This behaviour already results in the well known ant trails.
The AntWorld simulation works in rounds (similar to time-slices). Within each round, each ant makes one move. If an ant is not in carry mode and is on a node with food parts, it takes one piece of food and enters carrying mode. Note that it may still move within the current round. On the other hand, if an ant carries some food, it follows the links towards the inner circle one node per round. During its way home (towards the unique hill at the centre of all node), on each visited node (including the node that it picked food from) the ant drops 1024 parts of pheromones in order to guide other ants to the food place. However, if a carrying ant is on the hill, it drops the food and enters the search mode. It may leave the hill within the same round. Any ant without food is in search mode. While in this mode, the ant checks the neighbouring node(s) of the next outer circle for pheromones. If some hold more than 9 parts of pheromones, the ant chooses one of these nodes randomly. Otherwise, the ant moves to any of its neighbour nodes based on a fair random choice (but never enter the ant hill). Whenever during one round an ant is on a node on the outmost circle, a new circle of nodes shall be created. For each outmost grid node, a new grid node is created; but three nodes are created in the case of a main axis node. During the creation of this next circle, every 10^{th} node shall carry 100 food parts. If a circle has for example 28 nodes, node 10 and node 20 of that circle shall have food. Thus, this circle would need just two more nodes to create a third food place. Therefore, these 8 nodes are kept in mind and during

the creation of the next circle (in our example with 36 nodes) we add another food place when two more nodes have been added. Thus, across circles, every 10^{th} node becomes a food place. After each round, all pheromones shall evaporate: reducing by 5%. Also, the hill shall consume the food brought to it by creating one new ant per delivered food part.
To emphasize the advantages of the use of DEVS for controlled graph rewriting, the AntWorld case study has been extended as follows. The ants running around seeking for food are not protected from external factors. In fact, a human could step on an area discovered by the ants. Hence at random points in time, ants will be killed. This happens on a grid node chosen at random as well as its neighbouring nodes. Irrespective of whether ants are present or not on these four nodes, the nodes will lose food parts by a factor of 2 and pheromones by a factor of 10. Section 4 will show how this can be modelled by simulating the user decision of when a human step will occur and Section 5 how this can be done in real time, allowing an actual user to intervene.

2.1 The AntWorld Language (Abstract and Concrete Syntax)

As shown in Fig. 2, the AntWorld formalism consists of Ants and GridNodes. An *Ant* element can go on a *GridNode* which can also be a *Hill*. GridNodes can hold pheromones and food parts. Ants can be in "carry mode". The grid nodes are connected in circles centred at the hill in a very specific way. This is why the meta-model differentiates between connections in the same circle and to the next circle for neighbouring grid nodes. Furthermore, different strategies for the generation of the grid are used depending on whether it is a node along the two main axes. This is why we distinguish between main axis nodes and the other grid nodes. A node counter is also needed to decide which generated node will hold food parts. Using AToM^3 [5] as a modelling environment enables one to associate a concrete syntax for each meta-model element. This is depicted in Fig. 2 by the pictures attached to the elements by red dashed lines.

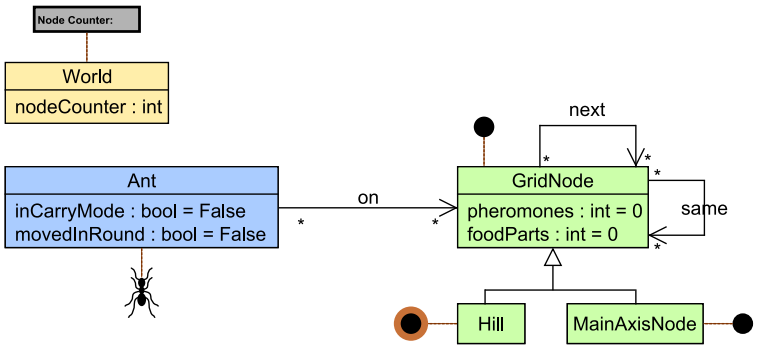


Fig. 2. AntWorld Meta-Model

2.2 The AntWorld Semantics (Graph Transformation)

As we have seen at the beginning of this section, the semantics of the AntWorld formalism is described in terms of simulation rules. In our case, these rules are graph rewriting rules taking as input a host graph (model) and producing as output the transformed graph. This encodes the state changes (*i.e.*, dynamics)

of the system. In MoTif, a rule consists of a LHS, a RHS, and optionally a NAC. The LHS represents a pre-condition pattern to be found in the host graph along with conditions on attributes. The RHS represents the post-condition after the rule has been applied on the matched subgraph by the LHS. The NAC represents what pattern condition in the host graph shall not be found, inhibiting the application of the rule if it is. Additionally, hints can be provided through pivot information. A rule using a pivot on a node of the LHS pattern binds the matching process to a previously specified matched node (this is useful for local search).

To illustrate how a graph rewriting rule is applied, we will consider the *ConnectNodesInSameCircle* rule from Fig. 3. This rule states that whenever two neighbouring nodes[1] N and M are on the same circle and are each linked to a *GridNode* (labelled respectively 1 and 2) on the next circle, a connection *toSameCircle* must be drawn between these latter nodes in the same direction N and M are connected. However, 1 may not be the source of a link to a node on the same circle nor 2 be the target of such a link.

Also, N must be bound to the pivot that this rule receives and M will be the pivot of the next rule. This internal dependency between rules is the essence of how local rule application is achieved in MoTif. For this particular case, the next rule that will be executed is again *ConnectNodesInSameCircle* as the control structure will show, in Section 3. This makes the connecting step between generated nodes ordered in clockwise direction.

One of the advantages of using graph transformation for transforming models is that they can easily be described in a visual language which makes the rules human readable. The remaining rules described in the appendix also use concrete syntax.

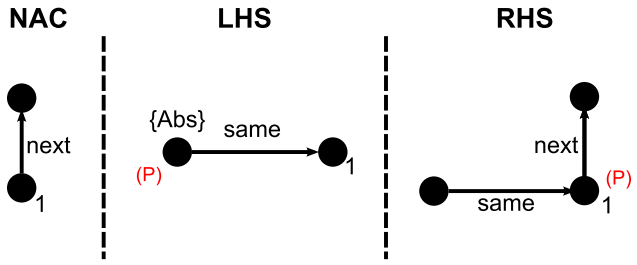


Fig. 3. The *ConnectNodesInSameCircle* rule

3 A Meta-Model for DEVS-based Graph Transformation

MoTif is a controlled graph transformation language. It offers a clean separation of the transformation entities (*i.e.*, the rewriting rules) from the structure and flow of execution of the transformation. While Section 2.2 outlined the graph transformation rules, we focus here on the structural and control flow aspect of MoTif. Revisiting Fig. 1 where up to now only the DEVS meta-model was mentioned, we will now see how a MoTif model is a DEVS model specialized for graph transformation.

The central elements of this DEVS-based graph transformation meta-model are the rule blocks. The graphs are encoded in the events that flow through the ports from block to block. The atomic block

[1] We use the term nodes in the general sense (*i.e.*, including *GridNode*, *MainAxisNode*, and *Hill*) because subtype matching can be used. This allows an pattern element to be matched to any element from the same class or from a sub-class of it. In MoTif, such a pattern element is flagged with the {Abs} label, short for abstract.

`ARule` (for "Atomic Rule") is the smallest transformation entity and the coupled block `CRule` (for "Coupled Rule") is meant for composition of rule blocks. A rewriting rule is part of the state of an *ARule* as a reference to the compiled rule. Rule application is performed in two phases: (1) the matching (where all the possible matches are found) and (2) the transformation on one or more matches. The *ARuleState* also keeps track of the graph and pivot received. The time advance of an *ARule* can be specified at modelling time to set its execution time (both match and transform). Otherwise the time advance is $+\infty$.

ARule blocks receive packets (a graph, with potentially a pivot node) from the *AGraphIn* port. In case of success (*i.e.*, when at least one match has been found), the packet containing the transformed graph is output through the *ASuccessOut* port. In case of failure, the original graph is sent through the *AFailOut* port. Furthermore, it is possible to enable pivot passing for these two outports. For the success outport, either the new pivot specified by the rewriting rule or the original received pivot is passed on to the next block. In the case of multiple matches found in the received graph, a host graph ever received by the *ANextIn* port will only apply the transformation on the next match without running the matching phase one more time. This feature is very useful as we will see in the AntWorld example for the flow logic and performance. On reception of an event through the *AResetIn* port, the rule application is cancelled and the state of the *ARule* is reset. Similar ports are available for a *CRule* block which serve as interface from its incident blocks to its sub-models.

To increase the expressiveness of the language MoTif, additional rule blocks have been added. Among them is the *FRule* which will be used in our example. It is an *ARule* that applies its transformation phase to *all* the matches found (in arbitrary order) before sending the new graph. The matches are assumed to be parallel independent.

As in a general purpose DEVS model, atomic and coupled rule blocks are connected through their ports. There could be one-to-many or many-to-one connections between them. The semantics of an *(A/C)SuccessOut* outport[2] connected to many *(A/C)GraphIn* inports is the parallel execution of the rules encoded in the receiving blocks. Since classical DEVS is used here, the parallel execution of the external transition of these rule blocks is serialised as specified by the `select` function. In our case, one block is chosen at random, first among the matching rules and then among those that failed. Many-to-one connections between rule blocks ports are not encouraged, since different graphs will be received by a single *ARule* at the same time.

In graph grammars, it is sometimes wished to have many rules that match but let only one execute. That is why MoTif introduces the `Selector` block. Such a pattern can be found in Fig. 6 invloving, for example, a `Selector` and the `Generate` and `CreateFood` *ARules* connected to it. Its purpose is to receive, through its *ASuccessIn* inport, the transformed graph sent from an *ARule* that has been chosen by the `select` function. Instantaneously it outputs an event via its *AResetOut* outport, forcing all remaining rules to reset their state. Then, with a time advance of 0, the *Selector* passes the packet it received to the next block(s) via its *ASuccessOut* port. In case of failure of all *ARules*, the rule selected by the `select` function sends its original packet to the *AFailIn* inport of the *Selector*. In return, the *Selector* forces the reset of all these rules and outputs the packet received through its *AFailOut* port.

Since the semantics of DEVS is the compositional parallelization of atomic blocks, MoTif allows rules to conceptually be applied in parallel. This leads to what we call "threads" of rule applications, *e.g.*, the *HumanStep CRule* has four such threads in Fig. 7(c) (which will be described later in the context of the case-study). Therefore a *Synchronizer* is needed to merge and synchronize the concurrent threads. Our approach uses *in-place* transformation of models, which means that the events sent and received are references to the host model, in contrast with *out-place* transformation where rules work on copies of the host model. This avoids the undecidable problem of merging transformed models. In that sense, the *Synchronizer* waits until all the threads have sent their packets through its *ASuccessIn* and/or

[2] The *(A/C)FailOut* has an analogous semantics.

AFailIn inports. Only then will it send the transformed graph through its *ASuccessOut* port if at least one thread has succeeded or else it will send the unmodified graph through its *AFailOut* port.
To formalise these concepts, we define each of these elements in terms of a DEVS structure.

3.1 The *ARule*

The *ARule* is an *atomicDEVS*, parameterised by a rule $\mathtt{r}$ and by σ_1 to determine if a pivot is sent on a successful matching and σ_2 on failure.

$$ARule_{\mathtt{r},\sigma_1,\sigma_2} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

where

$$\begin{aligned}
T &= \mathbb{R}_0^+ \\
S &= \{s = (\gamma, \rho, \epsilon, \sigma_1, \sigma_2, \Sigma(\mathtt{r})) \,|\gamma \in G, \rho \in V_G \cup \{\phi\}, \epsilon, \sigma_1, \sigma_2 \in \{\mathtt{true},\ \mathtt{false}\}\} \\
\tau(s) &= \begin{cases} t \in [0,\infty) & \text{if } \epsilon = \mathtt{true} \\ \infty & \text{otherwise} \end{cases}, \forall s \in S \\
X &= X_{\mathtt{AGraphIn}} \times X_{\mathtt{AResetIn}} \times X_{\mathtt{ANextIn}} \\
X_{\mathtt{AGraphIn}} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\
X_{\mathtt{AResetIn}} &= \{\mathtt{false}\} \cup \{\phi\} \\
X_{\mathtt{ANextIn}} &= \{\langle \gamma, \rho \rangle_n\} \cup \{\phi\} \\
Y &= Y_{\mathtt{ASuccessOut}} \times Y_{\mathtt{AFailOut}} \\
Y_{\mathtt{ASuccessOut}} &= \{\langle \gamma', \rho' \rangle\} \cup \{\phi\} \\
Y_{\mathtt{AFailOut}} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\
\omega &: T \to X \\
\delta_{int}(\gamma, \rho, \epsilon, \sigma_1, \sigma_2, \Sigma(\mathtt{r})) &= (\gamma, \rho, \mathtt{false}, \sigma_1, \sigma_2, \Sigma(\mathtt{r})) \\
\delta_{ext}((s,e), x) &= \begin{cases} (\gamma, \rho, \mathtt{true}, \sigma_1, \sigma_2, \Sigma^M(\mathtt{r})) & \text{if } x = \langle \gamma, \rho \rangle \\ (\gamma_n, \rho, \mathtt{true}, \sigma_1, \sigma_2, \Sigma^M(\mathtt{r})) & \text{if } x = \langle \gamma, \rho \rangle_n \\ (\gamma, \rho, \mathtt{false}, \sigma_1, \sigma_2, \Sigma^i(\mathtt{r})) & \text{if } x = \mathtt{false} \end{cases} \\
\lambda(s) &= \begin{cases} \langle \gamma', \rho' \rangle & \text{if } \sigma_1 = \mathtt{true} \wedge \Sigma(\mathtt{r}).\mathtt{INSTATE}(\Sigma^T(\mathtt{r})) \\ \langle \gamma', \phi \rangle & \text{if } \sigma_1 = \mathtt{false} \wedge \Sigma(\mathtt{r}).\mathtt{INSTATE}(\Sigma^T(\mathtt{r})) \\ \langle \gamma', \rho \rangle & \text{if } \sigma_2 = \mathtt{true} \wedge \Sigma(\mathtt{r}).\mathtt{INSTATE}(\Sigma^\perp(\mathtt{r})) \\ \langle \gamma', \phi \rangle & \text{if } \sigma_2 = \mathtt{false} \wedge \Sigma(\mathtt{r}).\mathtt{INSTATE}(\Sigma^\perp(\mathtt{r})) \end{cases}
\end{aligned}$$

In this notation, γ is a graph from the set of all possible graphs G. ρ is a node of this graph representing the pivot. $\langle \gamma', \rho' \rangle$ is the resulting packet after the transformation phase of the application of $\mathtt{r}$. Note how it is possible to have $\rho' = \rho$. ϵ is used to determine if the *ARule* is active. The function Σ gives the state of the rule $\mathtt{r}$. It is illustrated by the automaton in Fig. 4. Σ^M gives the state of $\mathtt{r}$ after the matching phase, Σ^T gives the state of $\mathtt{r}$ when it has at least one unprocessed match left and $\Sigma^\perp$ gives the state of $\mathtt{r}$ when there are none.

3.2 The *CRule*

The *CRule* is defined exactly like a *coupledDEVS*

$$CRule = \langle X, Y, N, \{M_i | i \in N\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

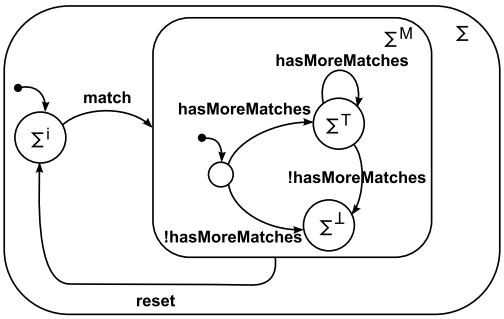


Fig. 4. The state automaton of a rule `r`

where

$$
\begin{aligned}
X &= X_{\texttt{CGraphIn}} \times X_{\texttt{CResetIn}} \times X_{\texttt{CNextIn}} \\
X_{\texttt{CGraphIn}} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\
X_{\texttt{CResetIn}} &= \{\texttt{false}\} \cup \{\phi\} \\
X_{\texttt{CNextIn}} &= \{\langle \gamma, \rho \rangle_n\} \cup \{\phi\} \\
Y &= Y_{\texttt{CSuccessOut}} \times Y_{\texttt{CFailOut}} \\
Y_{\texttt{CSuccessOut}} &= \{\langle \gamma', \rho' \rangle\} \cup \{\phi\} \\
Y_{\texttt{CFailOut}} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\}
\end{aligned}
$$

The *select* function is described by these prioritized algorithmic steps:

1. If the *Selector* is in the *imminent list*, choose the *Selector*.
2. Among all the rules that still have a match, choose a corresponding *ARule* from the *imminent list* at random, no matter what depth it is at inside this *CRule*.
3. At this point no rule has any unprocessed match left, then choose any of the *ARule* in the *imminent list*.
4. Finally, the *imminent list* contains either custom atomic blocks or *Synchronizers*. Proceed with a first-in-first-out selection.

3.3 The *Selector*

The *Selector* is also an *atomicDEVS*

$$Selector = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

where

$$
\begin{aligned}
T &= \mathbb{R}_0^+ \\
S &= \{s = (t, f, \langle\gamma, \rho\rangle) \,|t, f \in \{\texttt{true, false}\}, \gamma \in G \cup \{\phi\}, \rho \in V_G \cup \{\phi\}\} \\
\tau(s) &= \begin{cases} 0 & \text{if } t = \texttt{true} \vee f = \texttt{true} \\ \infty & \text{otherwise} \end{cases}, \forall s \in S \\
X &= X_{\texttt{ASuccessIn}} \times X_{\texttt{AFailIn}} \\
X_{\texttt{ASuccessIn}} &= \left\{\left\langle\gamma', \rho'\right\rangle\right\} \cup \{\phi\} \\
X_{\texttt{AFailIn}} &= \{\langle\gamma, \rho\rangle\} \cup \{\phi\} \\
Y &= Y_{\texttt{ASuccessOut}} \times Y_{\texttt{AFailOut}} \times Y_{\texttt{AResetOut}} \\
Y_{\texttt{ASuccessOut}} &= \left\{\left\langle\gamma', \rho'\right\rangle\right\} \cup \{\phi\} \\
Y_{\texttt{AFailOut}} &= \{\langle\gamma, \rho\rangle\} \cup \{\phi\} \\
Y_{\texttt{AResetOut}} &= \{\texttt{false}\} \cup \{\phi\} \\
\omega &: T \to X \\
\delta_{int}(s) &= (\texttt{false}, \texttt{false}, \langle\phi, \phi\rangle) \\
\delta_{ext}((s, e), x) &= \begin{cases} (\texttt{true}, \texttt{false}, \langle\gamma', \rho'\rangle) & \text{if } x = \langle\gamma', \rho'\rangle \\ (\texttt{false}, \texttt{true}, \langle\gamma, \rho\rangle) & \text{if } x = \langle\gamma, \rho\rangle \end{cases} \\
\lambda(s) &= \begin{cases} \{\langle\gamma', \rho'\rangle\} \cup \{\texttt{false}\} & \text{if } s = (\texttt{true}, \texttt{false}) \\ \{\langle\gamma, \rho\rangle\} \cup \{\texttt{false}\} & \text{if } s = (\texttt{false}, \texttt{true}) \end{cases}
\end{aligned}
$$

3.4 The *Synchronizer*

The *Synchronizer* is also an *atomicDEVS*, parametrized by the number of threads θ to synchronize.

$$Synchronizer_\theta = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \tau\rangle$$

where

$$
\begin{aligned}
T &= \mathbb{R}_0^+ \\
S &= \{s = (t, f, \theta, \langle \gamma, \rho \rangle) \,|t, f, \theta \in \mathbb{N}, \gamma \in G \cup \{\phi\}, \rho \in V_G \cup \{\phi\}\} \\
\tau(s) &= \begin{cases} 0 & \text{if } t + f = \theta \\ \infty & \text{otherwise} \end{cases}, \forall s \in S \\
X &= X_{\texttt{ASuccessIn}} \times X_{\texttt{AFailIn}} \\
X_{\texttt{ASuccessIn}} &= \left\{\left\langle \gamma', \rho' \right\rangle\right\} \cup \{\phi\} \\
X_{\texttt{AFailIn}} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\
Y &= Y_{\texttt{ASuccessOut}} \times Y_{\texttt{AFailOut}} \times Y_{\texttt{AResetOut}} \\
Y_{\texttt{ASuccessOut}} &= \left\{\left\langle \gamma', \rho' \right\rangle\right\} \cup \{\phi\} \\
Y_{\texttt{AFailOut}} &= \{\langle \gamma, \rho \rangle\} \cup \{\phi\} \\
\omega &: T \to X \\
\delta_{int}(s) &= (0, 0, \theta, \langle \phi, \phi \rangle) \\
\delta_{ext}((s, e), x) &= \begin{cases} (t + 1, f, \theta, \langle \gamma', \rho' \rangle) & \text{if } x = \langle \gamma', \rho' \rangle \\ (t, f + 1, \theta, \langle \gamma, \rho \rangle_{-1}) & \text{if } x = \langle \gamma, \rho \rangle \end{cases} \\
\lambda(s) &= \begin{cases} \{\langle \gamma', \rho' \rangle\} \text{ if } t \geq 1 \\ \{\langle \gamma, \rho \rangle\} \quad \text{otherwise} \end{cases}
\end{aligned}
$$

On top of these constructs, pure atomic and coupled DEVS models are also allowed to be present in MoTif models. This allows the modeller to add customised behaviour to the transformation model. *De facto*, the MoTif code generator makes use of it, when compiling the model down to an executable model transformation environment, by modelling the user of the transformation as well as an interface between the rule model and the user.

4 Using MoTif for the AntWorld Simulator Case Study

Having described the DEVS-based transformation language, we will now explore how MoTif models can be used. MoTif is a meta-modelled language and is provided with graphical visual concrete syntax. Figure 5 shows the overall structure of the DEVS model for the AntWorld graph transformation. Each block is shown with its ports along with the connections. The execution of the transformation is triggered by some user control. User intervention (such as a possible interrupt of a running simulation) is modelled in the *User* block, since the DEVS formalism allows one to specify pre-emptive external 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 management of the transformation steps. The two rule blocks, *Round* and *HumanStep*, can both receive the host graph from the *Controller* and return a graph, transformed or not. This approach was also used to model a PacMan game [6].
Figures 6 and 7 show the core of the transformation model. The top left triangle on each rule block represents the *GraphIn* port. On the top right, the triangle with a line through it is the *ResetIn* port. The two small filled triangles on the left represent the *NextIn* port. At the bottom left, the double-lined triangle is the *SuccessOut* port and at the bottom right, the filled double-lined port is *FailOut*. Pivot passing is enabled when there is a round at the summit of one of the triangular outports. For the *Selector* and the *Synchronizer*, the thick-lined triangle is the *SuccessIn* port and the filled triangle is the *FailIn* port. On the top of the *Selector*, the triangle with a line through it pointing up is the *ResetOut* port.

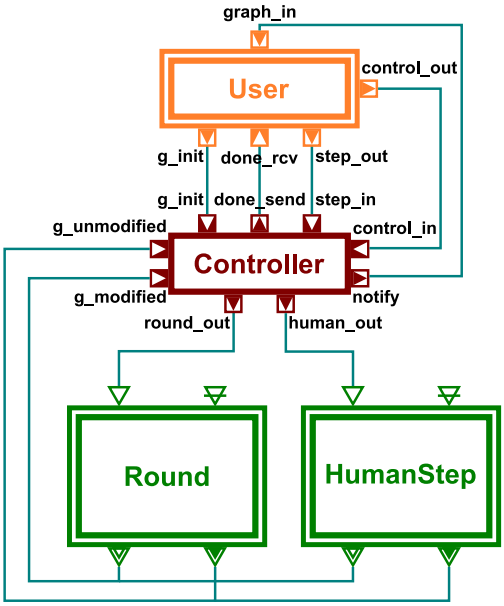


Fig. 5. The overall transformation model in MoTif notation

For completeness, Table 1 lists all the rules with a brief description used for a solution to the extended AntWorld case study using MoTif.

4.1 The Round Block

The AntWorld simulation is run in rounds. Figure 6(a) illustrates how a round is layered to first run the *AntMovement* sub-transformation (Fig. 7(a)), then when no more rule in this *CRule* is applicable, run the *GenerateCircle* (Fig. 6(b)) sub-transformation, and finally run the *EndOfRound* (Fig. 7(b)) sub-transformation.

At the level of the *AntMovement CRule*, the first rule looks for an ant to grab a food part. Having already explained in detail the description of the *ConnectNodes* rule, the reader is referred to the appendix for a description of all the rules.. The name of every *ARule* block matches the name of the rule itself. If the *GrabFood* rule succeeds this same ant moves one step towards the hill. Then the graph is sent back to the *GrabFood ARule* but via the *ANextIn* port to only choose the next matching and apply the transformation. This loop continues till no more ants can move towards the hill. Afterwards, an ant found in carry mode on the hill drops its food part and goes into search mode. When in search mode, the ant first tries to follow a pheromone trail or else moves randomly in any possible direction, either on a neighbouring node on the same circle or on a neighbouring node in the previous or following circle. This search mode behaviour is achieved by the four *ARules* connected to the *Selector* which find their corresponding matches in parallel and only one is chosen randomly to apply its transformation. The scenario is repeated for every ant matched by the *DropFood* rule. When no ant is provided by *DropFood*, all (iteratively chosen in random fashion) ants in search mode which have not moved yet move to a neighbouring node.

The (possibly) new graph is passed onto the *GenerateCircle CRule* when there are no more ants left to be moved. A check looking for an outermost circle node reached by an ant is first verified. If no

	Rule	Description
1	*GrabFood*	When an ant is on a node with food parts, remove a food part, add some pheromones and put the ant becomes in carry mode.
2	*MoveTowardsHill*	When an ant is in carry mode and has not moved yet, make it move to the neighbouring node on the previous circle.
3	*DropFood*	When an ant is on the hill and is in carry mode, increase the food parts of the hill by one and put the ant in search mode and allowed to move.
4	*GoToPheromones*	If the bound ant is in search mode and has not moved yet and the neighbouring node on the next circle has more than nine pheromones, the ant moves to that node.
5	*GoToNextNodeOut*	If the bound ant is in search mode and has not moved yet, it moves to the neighbouring node on the next circle.
6	*GoToNextNodeIn*	If the bound ant is in search mode and has not moved yet, it moves to the neighbouring node on the previous circle, if it is not a hill.
7	*GoToSameNodeOut*	If the bound ant is in search mode and has not moved yet, it moves to the neighbouring node on the same circle, in the direction of the link between the two nodes.
8	*GoToSameNodeIn*	If the bound ant in search mode and has not moved yet is on a node, it moves to the neighbouring node on the same circle, in the opposite direction of the link between the two nodes.
9	*CheckOnOutCircle*	When an ant is on a node on the outmost circle, bind the node.
10	*GenerateMAN*	If the bound node is linked to a main axis node in the same circle and the latter node has no neighbour in the next circle, create three neighbours on the next circle linked to that node: the central new node being another main axis node. Bind the former main axis node.
11	*GenerateGN*	If the bound node is linked to a grid node node in the same circle and the latter node has no neighbour in the next circle, create a neighbour on the next circle linked to that node. Bind the former grid node.
12	*CreateFoodMAN1*	If a grid node is linked to a main axis node that is the neighbour on the next circle of the bound main axis node and the node counter is 9, then that grid node is made to hold 100 food parts and the node counter is increased by 3 (modulo 10).
13	*CreateFoodMAN2*	If a main axis node is the neighbour on the next circle of the bound main axis node and the node counter is 8, then that main axis node is made to hold 100 food parts and the node counter is increased by 3 (modulo 10).
14	*CreateFoodMAN3*	If a grid node is linked to a main axis node that is the neighbour on the next circle of the bound main axis node and the node counter is 7, then that grid node holds 100 food parts and the node counter is increased by 3 (modulo 10).
15	*CreateFoodGN*	If a grid node is the neighbour on the next circle of the bound grid node and the node counter is 9, then that grid node holds 100 food parts and the node counter is increased by 1 (modulo 10).
16	*UpdateNodeCtrMAN*	If a bound main axis node is found, then increase the node counter by 3 (modulo 10).
17	*UpdateNodeCtrGN*	If a grid axis node is found, then increase the node counter by 1 (modulo 10).
18	*ConnectNodes*	*cf.* Section 2.2
19	*AntBirth*	When the hill has some food part, remove one food part and create an ant in search mode that has not moved yet.
20	*EvaporatePheromones*	When a node has some pheromones multiply the number of pheromones by 0.95, rounding to the next integer value.
21	*Reset*	When an ant has already moved, change it to not moved.
22	*WeakenNode*	When a node is found, divide its food parts by 2 and its pheromones by 10. Bind this node.
23	*KillOnNode*	If an ant is found on the bound node, delete the ant and keep the binding.
24	*WeakenSameOut*	When a node is linked to the bound node in the same circle, divide its food parts by 2 and its pheromones by 10. Keep the binding.
25	*KillSameOut*	If an ant is found on a node linked to the bound node in the same circle, delete the ant and keep the binding.
26	*WeakenSameIn*	When a node is linked to the bound node in the same circle, divide its food parts by 2 and its pheromones by 10. Keep the binding.
27	*KillSameIn*	If an ant is found on a node linked to the bound node in the same circle, delete the ant and keep the binding.
28	*WeakenNextOut*	When a node is linked to the bound node in the next circle, divide its food parts by 2 and its pheromones by 10. Keep the binding.
29	*KillNextOut*	If an ant is found on a node linked to the bound node in the next circle, delete the ant and keep the binding.
30	*WeakenNextIn*	When a node is linked to the bound node in the previous circle, divide its food parts by 2 and its pheromones by 10. Keep the binding.
31	*KillNextIn*	If an ant is found on a node linked to the bound node in the previous circle, delete the ant and keep the binding.

Table 1. The different rules for the extended AntWorld case study

node fulfils this criterion, the graph is passed onto the *EndOfRound CRule*. Otherwise, the generation of nodes of the next circle in clockwise order is engaged. The creation of nodes starts at the node found by the *CheckOnOutCircle* rule. The transformation model takes care of the case where the creation happens at the level of a main axis node (three nodes are created, the middle one being also a main axis node) or a default grid node (only a default grid node is created). The four *CreateFood ARules* handle the specification that some food is placed on every tenth new node, the count of nodes being tracked by the *World* element. Once all nodes on the new circle are created, they get connected through *ConnectNodes*.
Finally, the end of the round is reached. For each food part on the hill one ant element is created; this is depicted by the *AntBirth*, *Selector* pair which makes the rule execute "for as long as possible". Note how *EvaporatePheromones* and *Reset* are all *FRules*, denoted by the "F" in the *ARule* box. This forces the rule to transform all its matches before outputting its packet. Note that this is safe, since no two matchings can be critical pairs. Subsequently all pheromones are evaporated and a final clean up is made. The order of these three *ARules* is arbitrary and could, in principle, be executed in parallel.

4.2 The HumanStep Block

The *HumanStep* semantics can be summarised by two actions: weakening and killing. Weakening a node reduces its food parts and pheromones, if any. Killing removes all ants on a node. When the *CRule* receives a graph, first an arbitrary node is chosen and the two actions are applied to it. *WeakenNode* sends *KillOnNode* the node it has chosen along with the transformed graph to apply the killing rule on this same node. Since all ants on the node are deleted, *KillOnNode* is an *FRule*. The same logic is repeated for each of the four neighbouring nodes. In this case, every kill and weaken *ARule* pair on neighbours can also be executed in parallel since the rules are parallel independent. As stated by the Local Church-Russer theorem [7], two rules are said to be parallel independent if the matching elements that overlap between the two LHS are preserved by each of the rules application. Proving the parallel independence of eight rules[3] is not the focus of this chapter and is therefore not discussed any further.

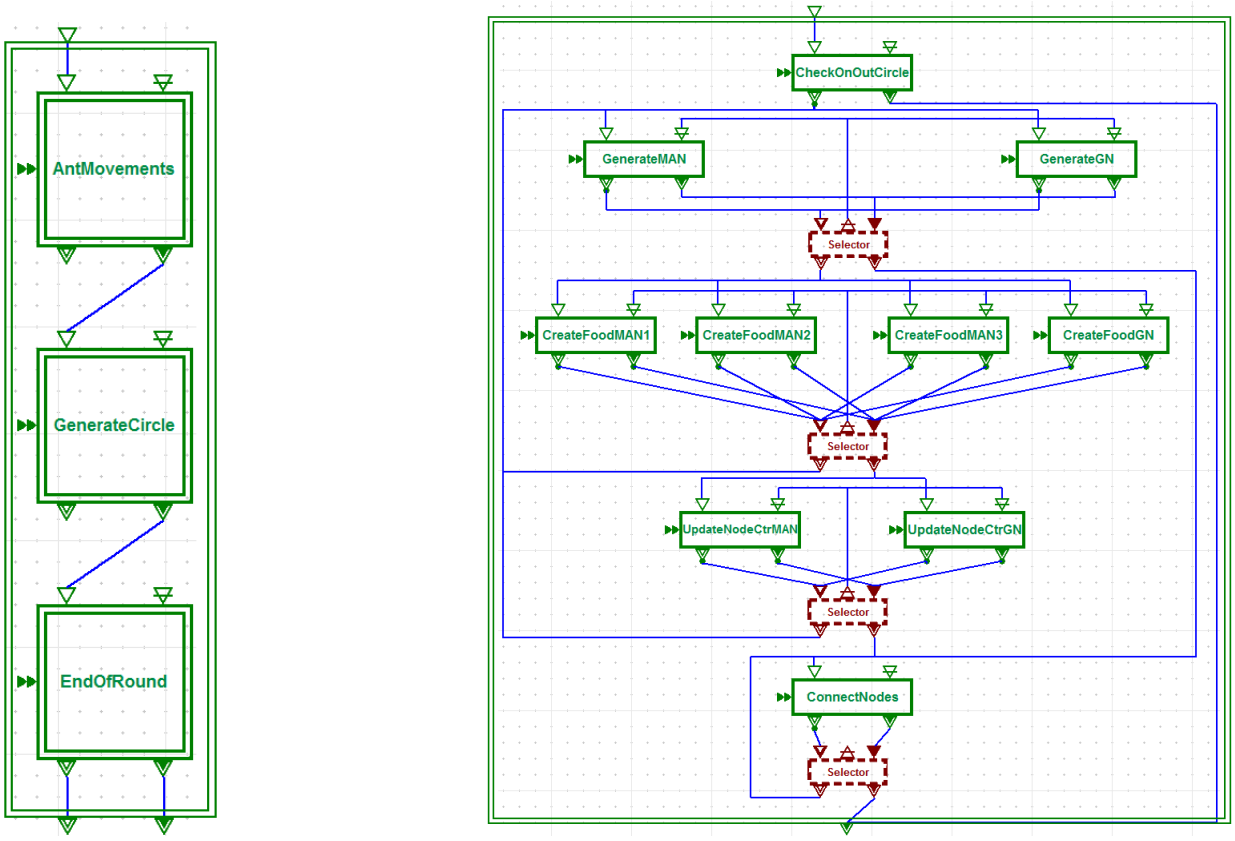
4.3 The Controller Block

The *Controller* atomic DEVS 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 *Round* sub-transformation model and waits for a graph in return. The returned graph may or may not be modified. However this cycle is interrupted when it receives an event from its *control_in* port. It will send the graph to the *HumanStep* sub-transformation model after *Round* has returned a graph. Every time a graph is received back, the *Controller* will notify the user by sending it the graph it just received; which happens to be after each round. This is repeated depending on the "steps" requests received. Note that the system could in principle receive multiple graphs at any one time (thanks to the data flow nature of DEVS), but we restricted it to a control flow in our case. Also, the user could request more "steps" even when there are some steps left in the running transformation.

4.4 The User Block

User is a coupled DEVS block that sends graphs and "steps" control signals and receives termination events. The graphs are Abstract Syntax Graphs (AToM^3's internal representation of models) of models

[3] The four pairs $WeakenSameOut, KillSameOut, WeakenSameIn, KillSameIn, WeakenNextOut, KillNextOut$, and $WeakenNextIn, KillNextIn$ are parallel independent.



(a) (b)

Fig. 6. Sub-models of the transformation model: the *Round CRule* in (a) and the *GenerateCircle* in (b) *CRule*

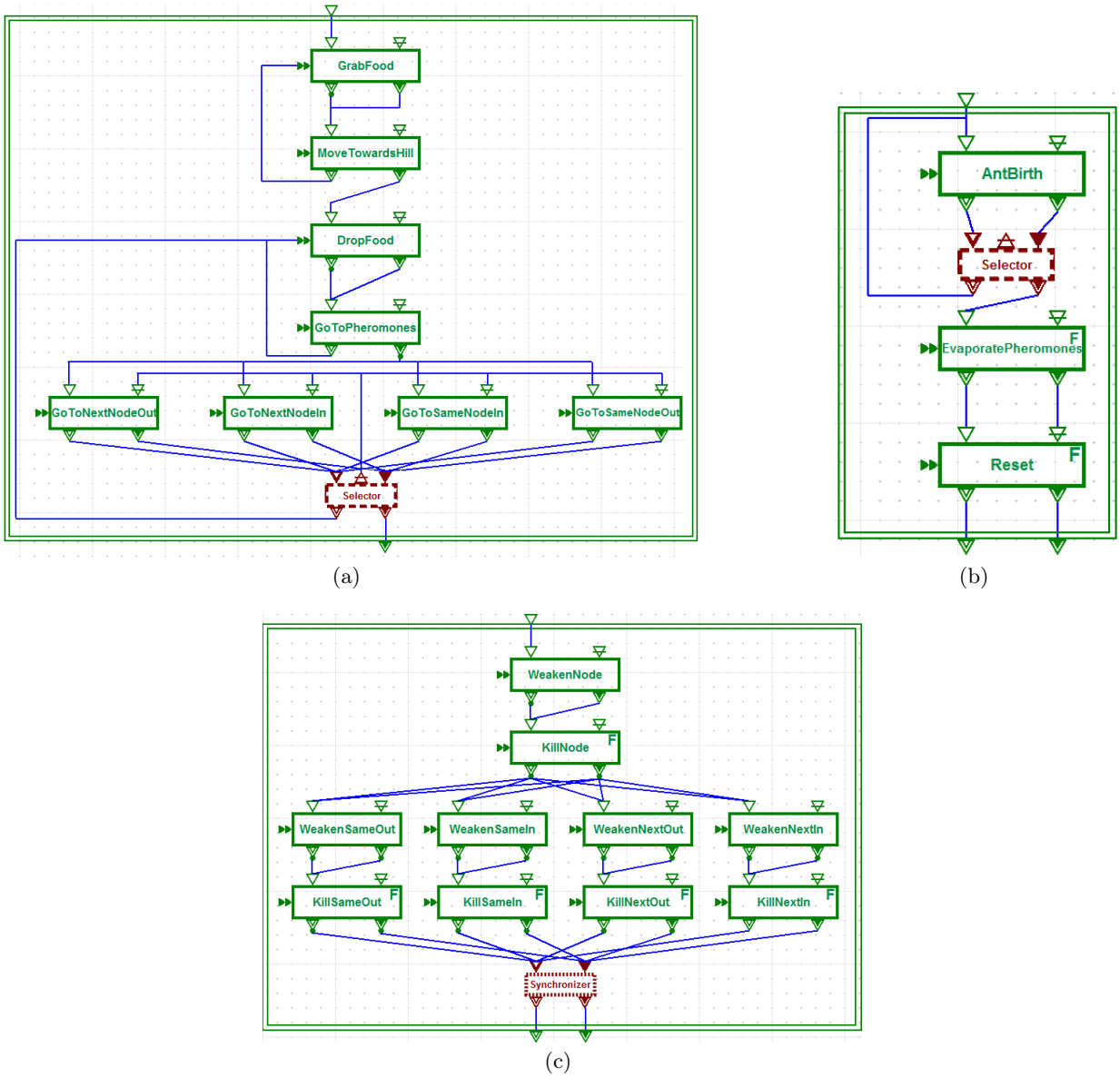


Fig. 7. Sub-models of the transformation model: the *AntMovements CRule* in (a), the *EndOfRound* in (b) *CRule*, and the *HumanStep* in (c) *CRule*

in the AntWorld language. Steps represent the number of iteration cycles 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). For our case study we let the simulation run "as long as possible". 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 *User* block are connected to the *Controller* block only. The *User* block is composed of two atomic sub-models: *UserInput* and *UserBehaviour*. The *User* is separated into two sub-models to distinguish the decision making of performing the *HumanStep* transformation from the interaction with the transformation system. Hence the *UserBehaviour*'s time advance is randomized to emulate the random time aspect of the human stepping on a node. This separation of the user is the key for extending standard graph transformations to event-driven timed graph transformations. Note how this event-based selection of rules is different from "Event-driven Graph Rewriting", as introduced in [8]. The authors [8] let the rule itself determine how to behave given an event. Hence it is the rule that "parses" the event. In our approach, the event is "parsed" by a separate atomic DEVS block and the appropriate rule is applied accordingly. This approach is therefore more modular.

4.5 Simulation Results

For the simulation experiments, an initial model was used with the following setup: 8 nodes, 1 hill, and 1 node counter and no ants. Figure 8 shows a snapshot of the model being transformed.
Some performance measurements have been collected at the end of each round. Table 2 shows some results per round, the number of circles present on the grid, the total number of nodes, the number of food parts present on the grid, and the total number of ants alive. Furthermore, for each round we show how long the transformation step took in seconds. So the 165^{th} round took 7 minutes and 18 seconds while the generation of the 13^{th} circle (100 nodes) took about 48 minutes. The total execution time was 23,890 seconds. Also, on average over the first 165 rounds the transformation time is about 144 seconds and without considering the time consumed by the *GenerateCycle* block, the average is 104 seconds. These measurements were taken on a Windows Vista machine with an Intel Core 2 Duo CPU with 1.5 GHz of RAM.

Round #	Circles	Nodes	Food	Ants	Time (sec)	Round #	Circles	Nodes	Food	Ants	Time (sec)
1	2	16	0	8	0	43	7	196	1724	34	17
8	4	64	500	8	0	44	7	196	1713	35	7
9	5	100	900	8	15	55	7	196	1651	46	10
10	5	100	900	8	1	56	8	256	2237	47	207
11	5	100	900	8	1	65	9	324	2891	56	48
12	5	100	900	8	1	73	10	400	3640	64	698
13	5	100	899	8	0	103	10	400	3423	94	74
14	6	144	1297	8	39	104	11	484	4214	95	1218
15	6	144	1297	8	0	105	12	576	5107	96	1924
32	6	144	1267	23	1	106	12	576	5099	97	117
33	6	144	1264	24	2	107	13	676	6093	98	2881
38	6	144	1243	29	3	165	13	676	5523	156	438

Table 2. Performance Measurements

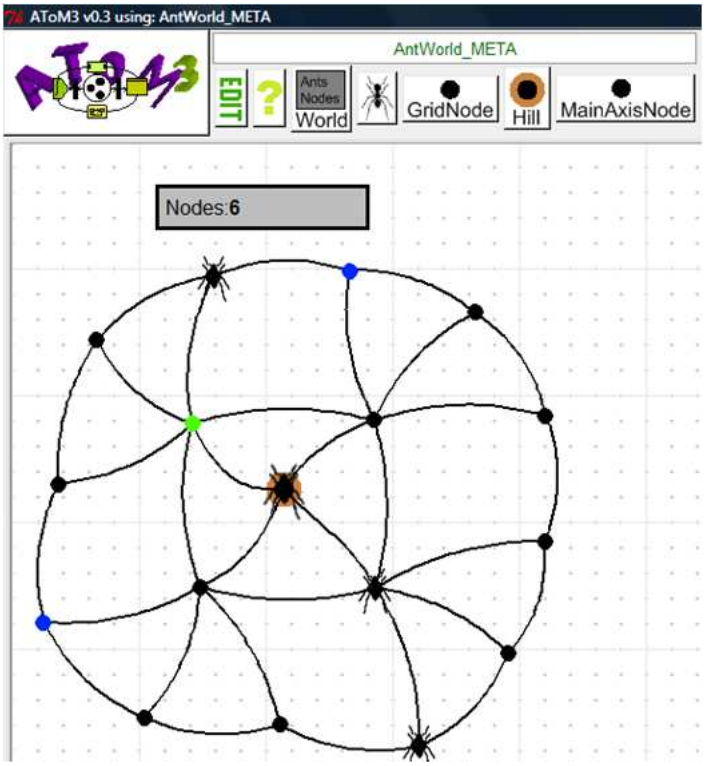


Fig. 8. Snapshot of the model in AToM^3 while being transformed

5 Timed Graph Transformation

In this section we briefly discuss the introduction of time in transformation languages and how our approach makes use of it.

5.1 Introduction of Time

DEVS is inherently a timed formalism. Hence, using DEVS as a semantic domain for graph transformation has a side effect of extending graph transformation with the notion of time.
Timed Graph Transformation, as proposed by Gyapay *et al.*, [9], integrates time in only one particular theoretical foundation of graph transformation: the double push-out approach [7]. They extend the definition of a graph transformation rule by introducing, in the model and rules, a "chronos" element that stores the notion of time. Rules can monotonically increase time.
In our approach, time is modelled and embedded both at the block entity level. In contrast with [9], it is the execution of a rule that can increase time and not the rule itself. This is done through the time advance of *ARules*. Hence, the control flow of the transformation has full access to it. As pointed out in [9], 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 than 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. To have more realistic simulations in the AntWorld example, we can give meaning to the time delay between the executions of different rules by modelling the user. For example, the ant movement rules may take more time than the generation of new circles and the rules at the end of round may take no time.

5.2 Real-Time Model Transformation and Deployment

Having control over time for simulation purposes can be extended to real-time applications. In [10] we have shown how using DEVS for programmed graph rewriting with time allows for simulation-based design. This was achieved on a game example where first the *UserBehaviour* block was enhanced with some artificial intelligence for path finding to optimize parameters such as the speed of the game. Then a web-based game was synthesised from the simulation model, where the *UserBehaviour* block was replaced by an actual human user. A real-time simulator, our Python implementation of RT-DEVS [11], was used.
A similar approach has been employed for the extended AntWorld case study. From the meta-model of the AntWorld formalism designed in AToM^3, an Ajax/SVG-based Web application was (mostly) synthesised (yet another model transformation). The web page consists of a visual representation of a given model. While the transformation runs, ants move and new node circles are created. Each node is equipped with an event handler listening to a mouse click from the user. This allows the user to "interrupt" the *Round* sub-transformation and trigger the *HumanStep* sub-transformation, passing the clicked node as initial pivot.

6 Related Graph Transformation Tools

Many graph transformation tools and languages have been developed during the past decade. Hence, we present those that describe a transformation in a controlled way (*i.e.*, programmed graph rewriting).
The Graph Rewriting And Transformation (`GReAT`) tool [12,13,14] treats the source model, the target model, and the temporary objects created during the transformation as a single graph using a unified

meta-model. 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 Object Constraint Language (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.

In the Visual Modelling and Transformation System (`VMTS`) [15,16], the LHS and RHS of a graph transformation rule are represented as two separate graphs. They can be linked (internal causality) by Extensible Stylesheet Language 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 [17]. 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 the 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.

The PROgrammed Graph REwriting System (`PROGReS`) [18] 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) [19,20], a completely redesigned graph transformation environment based on Java and UML. FUJABA's programmed graph rewriting system is based on Story Charts, an of Story Diagrams [20]. An activity in such a diagram contains either graph rewrite rules, which adopt a Collaboration Diagram-like representation [17], 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` [21] 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) [22]. These TGGs are subsequently compiled to Story Diagrams. This adds declarative power to FUJABA similar to that of the OMG's QVT (Query/View/Transformation – `www.omg.org`).

Although all these tools provide a control flow mechanism for graph transformations, many designed a new formalism for this purpose. Also, none of these exploit event-based transformations; MoTif not only allows that but the user and its interaction with the executing transformation can even be explicitly modelled, offering a user-centric approach to model transformations. Note that in the abovementioned tools, user-tool interaction is hard-coded. Furthermore, the notion of time is absent in these languages. Some do provide sophisticated, user friendly graphical interfaces while others are more efficient.

7 Conclusions

In this chapter, we have introduced the DEVS formalism as an appropriate semantic domain for "programmed" model transformation. 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. Each rule of a model transformation is encoded in an atomic-DEVS block (this is comparable to the atomicity of the rules in transformation tools such as PROGReS). The encoding is done automatically, by compiling declarative transformation rules into appropriate atomic-DEVS functions. Model transformation building blocks send and receive events through their output and input ports respectively. In those events, to-be-transformed graphs as well as optimization hints (such as pivot nodes in the tools GReAT and VMTS) are encapsulated.
Other events, related to information on the order in which rules are executed, are also fed to the channels (such as the event for resetting an *ARule* for example). The DEVS formalism is compositional: the behaviour of a DEVS block is independent of the context it is used in. This allows for modular re-use of building blocks and is one of the main reasons for choosing DEVS as a semantic domain for model transformation.
The use of DEVS allows for multi-level hierarchical modelling. 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). Parallel execution is provided by the DEVS formalism when an output port is connected to multiple input ports. If true parallelism is needed, the parallel DEVS [23] formalism can be used.
A side-effect of the use of DEVS as a semantic domain is the explicit introduction of the notion of time. This allows one to model a time-advance for every rule as well as to interrupt (pre-empt) rule execution.
The proposed approach was illustrated through the modelling of an extended version of the AntWorld model transformation benchmark. We showed how the use of DEVS ultimately allows for real-time simulation and execution.
Performance-wise, the generated code for individual transformation rule needs to be more efficient. Recent results from other transformation tools, such as VIATRA [24], indicate how higher performance may be achieved. Increasing the expressiveness of the rule pattern specification language is also ongoing work.

References

1. 2008/07/21. [Online]. Available: http://www.fots.ua.ac.be/events/grabats2008/
2. B. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, 1984.
3. J.-S. Bolduc and H. Vangheluwe, "The modelling and simulation package pythonDEVS for classical hierarchical DEVS," McGill University, MSDL Technical Report MSDL-TR-2001–01, June 2001.
4. E. Syriani and H. Vangheluwe, "Using MoTif for the AntWorld simulator case study," in *GraBaTs 2008 Tool Contest*, P. Van Gorp and A. Rensink, Eds., 2008.

5. J. de Lara and H. Vangheluwe, "AToM^3: A tool for multi-formalism and meta-modelling," in *FASE'02*, ser. LNCS, R.-D. Kutsche and H. Weber, Eds., vol. 2306. Grenoble(France): Springer-Verlag, April 2002, pp. 174–188.
6. E. Syriani and H. Vangheluwe, "Programmed graph rewriting with DEVS," in *AGTIVE'07*, ser. LNCS, M. Nagl and A. Schürr, Eds. Springer-Verlag, October 2007.
7. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, *Handbook of graph grammars and computing by graph transformation, Volume 1: Foundations*, G. Rozenberg, Ed. World Scientific Publishing Co., Inc., 1997.
8. E. Guerra and J. de Lara, "Event-driven grammars: Relating abstract and concrete levels of visual languages," *SoSym*, vol. 6, no. 6, pp. 317–347, 2007.
9. S. Gyapay, R. Heckel, and D. Varró, "Graph transformation with time: Causality and logical clocks," in *Procedings of ICGT 2002: 1st International Conference on Graph Transformation*, ser. LNCS, vol. 2505. Barcelona(Spain): Springer-Verlag, October 2002, pp. 120–134.
10. E. Syriani and H. Vangheluwe, "Programmed graph rewriting with time for simulation-based design," in *ICMT'08*, ser. LNCS, A. Pierantonio, A. Vallecillo, J. Bézivin, and J. Gray, Eds., vol. 5063. Zürich(Switzerland): Springer-Verlag, July 2008, pp. 91–106.
11. J. S. Hong, H.-S. Song, T. G. Kim, and K. H. Park, "A real-time discrete event system specification formalism for seamless real-time software development," *DEDS*, vol. 7, pp. 355–375, 1997.
12. A. Agrawal, "Metamodel based model transformation language," in *OOPSLA'03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*. Anaheim(USA): ACM Press, 2003, pp. 386–387.
13. A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, and A. Vizhanyo, "The design of a language for model transformations," *SoSym*, vol. 5, no. 3, pp. 261–288, September 2005.
14. A. Vizhanyo, A. Agrawal, and F. Shi, "Towards generation of high-performance transformations," in *Proceedings of the Third International Conference on Generative Programming and Component Engineering*, ser. LNCS, G. Karsai and E. Visser, Eds., vol. 3286. Springer-Verlag, 2004, pp. 298–316.
15. L. Lengyel, T. Levendovszky, G. Mezei, and H. Charaf, "Control flow support in metamodel-based model transformation frameworks," in *EUROCON'05*. Belgrade(Serbia): IEEE, November 2005, pp. 595–598.
16. ——, "Model transformation with a visual control flow language," *IJCS*, vol. 1, no. 1, pp. 45–53, 2006.
17. Object Management Group, *Unified Modeling Language Superstructure*, February 2009.
18. D. Blostein and A. Schürr, "Computing with graphs and graph rewriting," *SPE*, vol. 9, no. 3, pp. 1–21, 1999.
19. U. Nickel, J. Niere, and A. Zündorf, "Tool demonstration: The FUJABA environment," in *ICSE'00*. Limerick(Ireland): ACM Press, June 2000, pp. 742–745.
20. T. Fischer, J. Niere, L. Turunski, and A. Zündorf, *Theory and Application of Graph Transformations*, ser. LNCS. Paderborn(Germany): Springer-Verlag, November 2000, vol. 1764, chapter Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java, pp. 296–309.
21. C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr, "MOFLON: A standard-compliant metamodeling framework with graph transformations," in *Model Driven Architecture - Foundations and Applications: Second European Conference*, ser. LNCS, A. Rensink and J. Warmer, Eds., vol. 4066. Springer-Verlag, 2006, pp. 361–375.
22. A. Schürr, "Specification of graph translators with triple graph grammars," in *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, ser. LNCS, G. Tinhofer, Ed., vol. 903. Heidelberg(Germany): Springer-Verlag, June 1994, pp. 151–163.
23. A. C.-H. Chow and B. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular modeling formalism and its distributed simulator," *TSCS*, vol. 13, pp. 55–67, 1996.

24. D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Science of Computer Programming*, vol. 68, no. 3, pp. 214–234, 2007.