Softw Syst Model
DOI 10.1007/s10270-011-0205-0

REGULAR PAPER

A modular timed graph transformation language for simulation-based design

Eugene Syriani · **Hans Vangheluwe**

Received: 30 April 2009 / Revised: 18 April 2011 / Accepted: 17 May 2011
© Springer-Verlag 2011

Abstract We introduce the MoTif (Modular Timed graph transformation) language, which allows one to elegantly model complex control structures for programmed graph transformation. These include modular construction, parallel composition, and a temporal dimension in addition to the usual transformation control structures. The first part of this contribution formally introduces MoTif and its semantics is based on the Discrete EVent system Specification (DEVS) formalism which allows for highly modular, hierarchical modelling of timed, reactive systems. In MoTif, graphs are embedded in events and individual transformation rules are embedded in atomic DEVS models. A side effect of the use of DEVS is the introduction of an explicit notion of time. This allows one to model a time-advance for every rule as well as to interrupt (pre-empt) rule execution. In the second part, we design a case study to show how the explicit notion of time allows for the simulation-based design of reactive systems such as modern computer games. We use the well-known game of PacMan as an example and model its dynamics in MoTif. This also allows the modelling of player behaviour, incorporating data about human players' behaviour, and reaction times. Thus, a model of both player and game is obtained which can be used to evaluate, through simulation, the playability of a game design. We propose a playability performance measure and change the value of some parameters of the PacMan game. For each variant of the game thus obtained, simulation yields a value for the quality of the game. This allows us to choose an "optimal" (from a playability point of view) game configuration. The user model is subsequently replaced by a visual interface to a real player, and the game model is executed using a real-time DEVS simulator.

Keywords Simulation · DEVS · Graph transformation

Communicated by Prof. Robert France.

E. Syriani (✉) · H. Vangheluwe
McGill University, Montréal, Canada
e-mail: esyria@cs.mcgill.ca

H. Vangheluwe
University of Antwerp, 2020 Antwerp, Belgium
e-mail: hv@cs.mcgill.ca

1 Introduction

In 1996, Blostein et al. [1] described some issues regarding the practical use of graph rewriting, at that time very sporadic. 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, scalability, 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 widespread use of graph transformations. During the past 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 language to be target (programming) language *neutral*. The explicit incorporation of *time* is rare in current transformation languages. The above requirements were summarized recently in [3]. Programmed (or structured) graph transformation is one of the keys to making graph

Published online: 07 June 2011

transformation scalable and hence industrially applicable. Tools such as Fujaba [2], GReAT [4], VMTS [5], PROGReS [6], and MOFLON [7] support programmed graph transformation. These tools mostly introduce their own control flow language.[1]

The main contribution of this paper is the re-use of a discrete-event modelling and simulation formalism, such as the Discrete Event system Specification (DEVS), to describe the scheduler of a model transformation. Since DEVS inherently allows to build hierarchical models, the transformation language becomes highly modular by re-using specific components of a transformation. Another side-effect of using DEVS is the explicit introduction notion of time in model transformations. This allows one to model a time-advance for every rule as well as to interrupt (pre-empt) rule execution. Thanks to this notion of time, we implement a simulation-based design of reactive systems such as modern computer games. More precisely, the dynamics of the game is entirely modelled in the model transformation language.

We show the advantages of our approach by means of the well-known PacMan game example, presented in Sect. 2. In Sect. 3 we formally introduce the Modular Timed graph transformation (MoTif) formalism as well as its semantics, based on the DEVS formalism. Subsequently, the suitability of MoTif for Modelling and Simulation-based Design is demonstrated. The Pacman game is entirely modelled in MoTif in Sect. 4, including the game semantics, the transformation environment as well as the player behaviour. The simulation and optimization experiments as well as the synthesis of a real-time Pacman web application are explained in Sect. 5. Section 6 compares MoTif with other existing controlled graph transformation tools and languages. Finally, in Sect. 7, we conclude with advantages of our approach as well as future work.

2 The PacMan formalism

In order to illustrate the power of MoTif in the context of simulation-based design, we use a simplified version of the well-known video game PacMan throughout this paper. The **PacMan** language syntax and semantics are inspired by Heckel's tutorial introduction to graph transformation [8]. In what follows, we first synthesize a PacMan-specific visual modelling environment in our tool AToM[3] (A Tool for Multi-formalism and Meta-Modelling) [9] by defining a meta-model of the PacMan language. Subsequently, we model the semantics of the PacMan language by means of graph transformation rules.

[1] Though Fujaba's Story Diagrams are heavily based on UML Activity Diagrams.

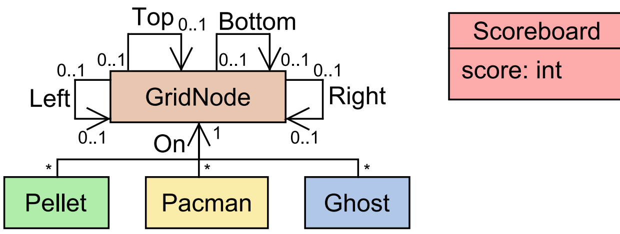


Fig. 1 The PacMan meta-model

2.1 The PacMan language (Abstract and concrete syntax)

The **PacMan** language has five distinct syntactic elements: PacMan, Ghost, Pellet, GridNode, and ScoreBoard. Figure 1 shows the meta-model (model of the abstract syntax) of this modelling language. PacMan, Ghost, and Pellet objects can be linked to GridNode objects with an On association. This represents that these objects can be "on" a grid node. The four self-associations Left, Right, Top, and Bottom between GridNode objects represent the geometric organization of the game area, similar to the classic PacMan video game. At a semantic level, these associations denote that PacMan and Ghost "may move" to a connected GridNode. A Scoreboard object holds an integer valued attribute score. For the concrete syntax of the **PacMan** language, an icon is associated with each of the meta-model's classes. For each of the meta-model's associations, a geometric/topological constraint relation is given. The On association between PacMan and Ghost entities for example is rendered as the PacMan icon being centred over the grid node icon.[2] Note that in this example, restrictions are applied neither on the number of instances of each meta-model class nor on the number of links to a GridNode instance.

2.2 The PacMan Semantics (Graph transformation)

The operational semantics of the **PacMan** formalism is defined by means of a collection of (graph) transformation rules. These rules take as input a host graph (model) and produce as output the transformed graph. The resulting graph may be only partly modified, e.g., the GridNode elements are preserved in the transformation. In MoTif, a rule consists of a left-hand side (LHS), a right-hand side (RHS), and optionally a negative application condition (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 shall not be found in the host graph, inhibiting the application of the rule if it is. In MoTif, the application

[2] Note that this is why links (instances of association) are not shown explicitly in Figs. 2–5.

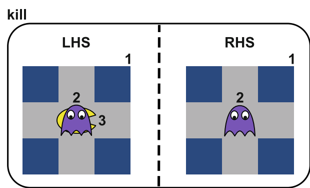


Fig. 2 PacMan Semantics: rule for Ghost killing PacMan

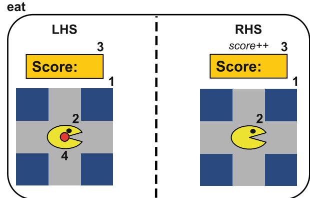


Fig. 3 PacMan Semantics: rule for PacMan eating Pellet

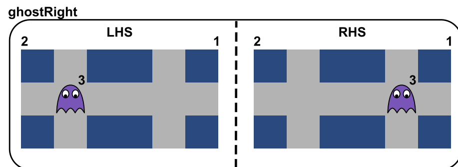


Fig. 4 PacMan Semantics: rule for Ghost moving right

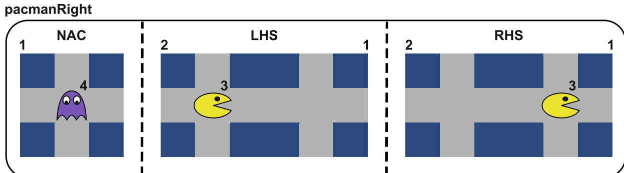


Fig. 5 PacMan Semantics: rule for PacMan moving right

of a rule follows the Single Push-Out approach [10]. Additionally, specific node binding through pivot[3] information provided to a rule can be used for local search.

Concrete visual syntax is used in the rules in Figs. 2–5. This feature of AToM[3] is particularly useful for domain-specific modelling. The **kill** rule in Fig. 2 shows killing: when a Ghost object is on a GridNode which has a PacMan object, that PacMan object is deleted. The **eat** rule in Fig. 3 shows eating: when a PacMan object is on a GridNode which has a Pellet object, the Pellet object is deleted and the score gets updated (using an attribute update expression). The **ghostRight** rule in Fig. 4 expresses the movement of a Ghost object to the right and the **pacmanRight** rule in Fig. 5 the movement of a PacMan object to the right. Note the presence of a NAC in the last rule which prevents the PacMan from moving to a GridNode which holds a Ghost (as this would imply certain death). Similar rules to move Ghost and PacMan objects up, down, and left are omitted.

[3] The term *pivot* refers to a similar concept in the GReAT language [4] for parameter passing: it represents an explicit reference to an element matched by a rule.

3 The MoTif language

The collection of rules described in the previous Section does not specify a rule application order. This Section introduces a control flow language which allows modellers to explicitly describe the order in which rules will be applied. This control flow language is based on the DEVS simulation formalism. In the following, it will be shown how the modularity and expressiveness of DEVS allow for elegant encapsulation of programmed model transformation, (i.e., graph rewriting) building blocks.

3.1 The discrete event system specification formalism

The DEVS formalism was introduced in the late seventies by Zeigler as a rigorous basis for the compositional modelling and simulation of discrete event systems [11]. It has been successfully applied to the design, performance analysis, and implementation of a plethora of complex systems such as peer-to-peer networks [12], transportation systems [13], and complex natural systems [14].

Figure 6 shows a possible 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 parallel composition of several DEVS sub-models which can be either atomic or coupled. Submodels have *ports*, which are connected by channels (represented here by the associations between the different ports). Ports are directional and are either Inport or Outport. The abstract classes (In/Out)port can be instantiated as an (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. 6. Note that the dynamic semantics of DEVS is not expressed by the meta-model. It will be informally presented hereafter.

An **atomic DEVS** model is a structure $\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$. S is a set of sequential **states**. 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 $\tau : S \rightarrow \mathbb{R}^+ \cup \{+\infty\}$, the **time-advance** function and $\lambda : S \rightarrow Y$, the **output function**. In this definition,

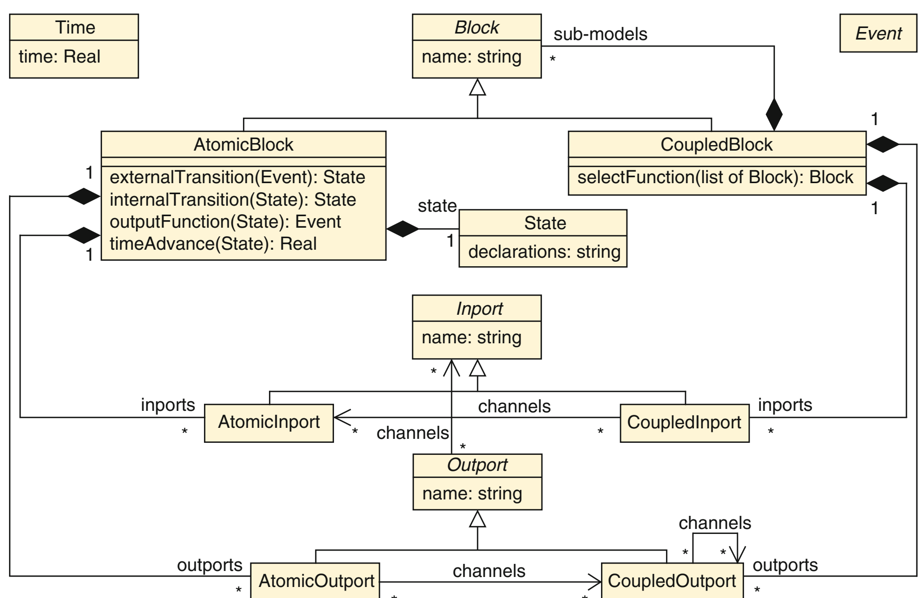
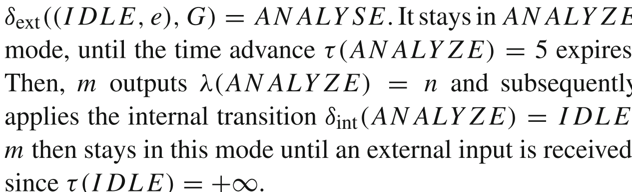


Fig. 6 The DEVS meta-model

$Q = \{(s, e)\,|s \in S, 0 \leq e \leq \tau(s)\}$ is called the **total state space**. For each $(s, e) \in Q$, e is called the **elapsed time**, the time the system has spent in a sequential state s since the last transition. The state of the atomic DEVS is initialized to $q_0 = (s_0, e_0)$, but in the sequel we only consider s_0 for simplicity. When the time is infinite, it is said to be *passivated*, and when it is zero, it is said to be *transient*.

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) produces an output event as specified by the output function. Then, it instantaneously jumps to a new state specified by the internal transition function. However, if an input event is received before the time for the next internal transition, then it is the *external transition* which is applied. The external transition depends on the current state, the time elapsed since the last transition, and the input event.

To illustrate the atomic DEVS concept, consider a user of a transformation system, who receives a graph G every time a rule is applied. Furthermore, after analysing the graph, he outputs a decision encoded as an integer $n \in \mathbb{N}$. We model the user's behaviour[4] by an atomic model m. It has two states $S = \{IDLE, ANALYZE\} \times \mathbb{N}$; the state is also used to store the computed integer. m can only receive a graph as input and hence $X = \{G\}$ and send an integer as output and hence $Y = \mathbb{N}$. S is initially in $IDLE$ mode. Upon reception of G, m applies the external transition


$\delta_{\text{ext}}((IDLE, e), G) = ANALYSE$. It stays in $ANALYZE$ mode, until the time advance $\tau(ANALYZE) = 5$ expires. Then, m outputs $\lambda(ANALYZE) = n$ and subsequently applies the internal transition $\delta_{\text{int}}(ANALYZE) = IDLE$. m then stays in this mode until an external input is received, since $\tau(IDLE) = +\infty$.

A **coupled DEVS** model named C is a structure $\langle X, Y, N, M, I, Z, \texttt{select}\rangle$ where X and Y are as before. N is a set of **component names** (or labels) such that $C \notin N$. $M = \{M_n | n \in N, M_n$ is a DEVS model (atomic or coupled) with input set X_n and output set $Y_n\}$ is a set of DEVS **sub-models** or **components**. $I = \{I_n \mid n \in N, I_n \subseteq N \cup \{C\}\}$ is a set of **influencer** sets for each component. $Z = \{Z_{i,n} | \forall n \in N, i \in I_n, Z_{i,n} : Y_i \to X_n \vee Z_{C,n} : X \to X_n \vee Z_{i,C} : Y_i \to Y\}$ is a set of **transfer functions** between connected components. $\texttt{select} : 2^N \to N$ 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 I_n of the components. 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 $(Z_{i,n})$ represent output-to-input translations between components. They 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

[4] This example is a simplification of the UserBehaviour model to be introduced in Sect. 4.2.

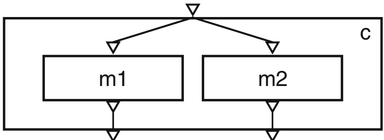


Fig. 7 A hierarchical DEVS model

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 at the same time. The modeller controls which of the conflicting sub-models undergoes its transition first by means of the `select` function.

To illustrate the coupled DEVS concept, we extend the previous example by involving different decision makers. It is visually depicted in Fig. 7. Suppose we now have two decision blocks m_1 and m_2, where m_i deterministically outputs i. The task is to output the computed numbers when a graph is received. For that, we construct a coupled model c where $X = \{G\}$ and $Y = \mathbb{N}$. We label the two inner models $M = \{m_1, m_2\}$ by $N = \{1, 2\}$, respectively. We then connect the inport of c to the inport of m_1 and the inport of m_2. We also connect the outports of both m_1 and m_2 to the outports of c. Therefore, $I = \{I_1 = \{c\}, I_2 = \{c\}, I_c = \{1, 2\}\}$. As for the transfer function, we define $Z_{c,1}(G) = Z_{c,2}(G) = G$ for the input-to-input channels and $Z_{1,c}(n) = Z_{2,c}(n) = n$ for the output-to-output channels. At simulation time (run-time), after G is received, m_1 and m_2 are scheduled to output and then perform their internal transition at the same time, since their time advance is 5 and they received the input at the same time. The `select` function then chooses which inner model will execute first, e.g., set `select`$(\{1, 2\}) = 1$.

We have developed our own DEVS simulator called `pythonDEVS` [15], grafted onto the object-oriented scripting language Python. We have used `pythonDEVS` for the work described in this paper.

3.2 Controlled graph transformation in DEVS

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 Sect. 2.2 outlined the graph transformation rules, we focus here on the structural and control flow aspects of MoTif. Figure 8 shows how a MoTif model is a DEVS model specialized for graph transformation by extending the meta-model in Fig. 6.

The central elements of this DEVS-based graph transformation meta-model are the rule blocks. The graphs to be transformed are embedded in the events that flow through the ports from block to block. The atomic block 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 (mentioned as ARuleState), as a reference to a compiled version of the rule (a function to be called whenever the rule needs to be invoked). Rule application is performed in two phases: (1) the *matching*, where all 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). If not set, the time advance is $+\infty$. Note that the time attribute of ARule parametrizes its time advance function.

ARules receive events from their AModelIn port (this is depicted by the inherited aggregation referring to inports). Henceforth, we will also refer to such events as "packets" as they have an internal structure, containing a graph and optionally a reference to a matched element, i.e., the pivot node. If a pivot is provided, the match of a rule is bound to that node. In case of success, (i.e., when at least one match has been found), an event containing the transformed graph is emitted 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 onto the connected block. In the case of multiple matches found in the received graph, a host graph received by the ANextIn port will only apply the transformation on the next match without running the matching phase again. This feature is very useful for complex flow logic and to increase performance. On reception of an event through the ACancelIn port, the rule application is cancelled. The cancel operation prohibits the rule from transforming its current match. Similar ports are available for a CRule block. They serve as interfaces from incident blocks to sub-models.

As in a general purpose DEVS model, atomic and coupled rule blocks are connected through their ports. One-to-many or many-to-one connections are possible. The semantics of an (A/C)SuccessOut outport[5] connected to many (A/C)ModelIn inports is the parallel execution of the rules encoded in the receiving blocks.

In the MoTif visual modelling language, the concrete syntax of an ARule is a single rectangle frame as illustrated in Fig. 9 for the Eat block. The top left triangle on a rule block represents the ModelIn port. On the top right, the triangle with a horizontal line through it is the CancelIn port. The two small filled triangles on the left represent the NextIn port. The bottom left tick symbol is the SuccessOut port and the bottom right "X" symbol is the FailOut port. Pivot passing is enabled when an outport is circled. A MoTif sub-model

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

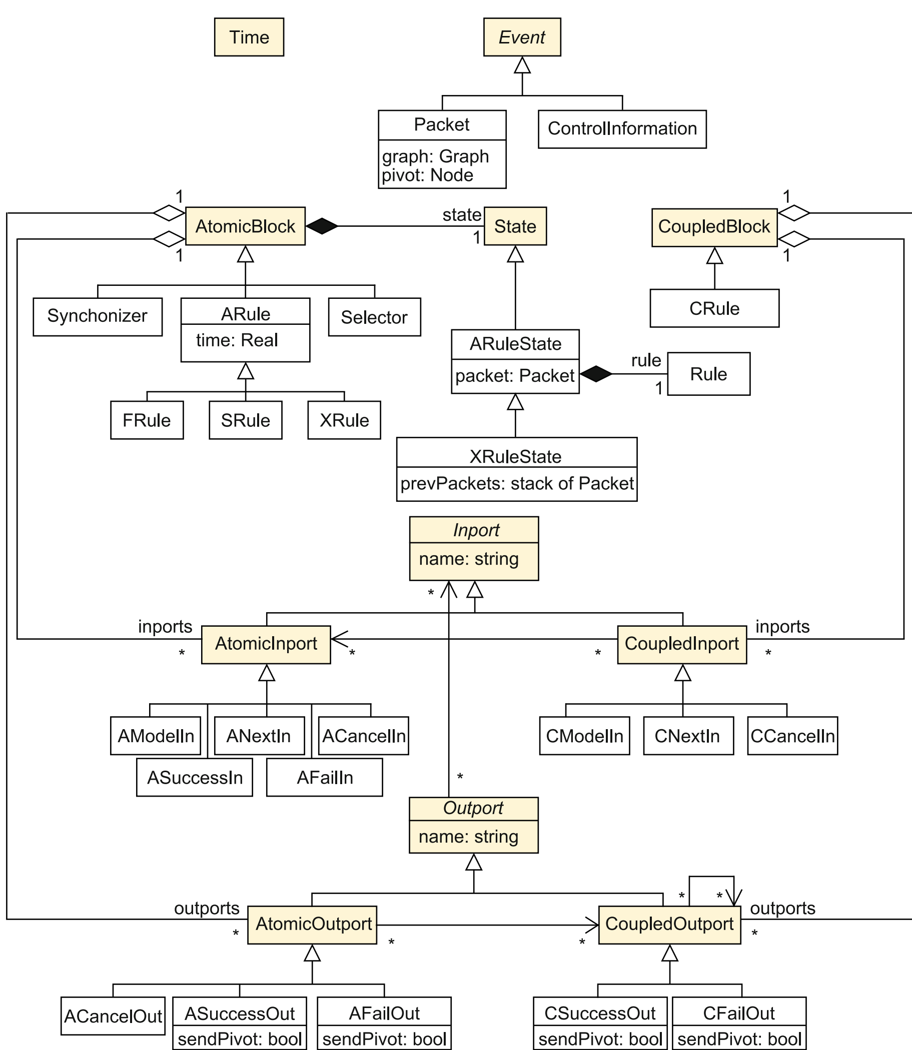


Fig. 8 The MoTif meta-model, based on the DEVS meta-model

encoding transformation rules can be part of a CRule. This is, for example the case in Figs. 9 and 10. CRules are visually depicted by a double rectangle frame. The same ports appear on both atomic and coupled transformation models which implies that they can be used interchangeably to thus modularly build complex hierarchical transformation models.

In graph transformation, it is sometimes desirable to non-deterministically select one rule to be applied when several other also match. That is why MoTif introduces the Selector block. 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 ACancelOut outport, forcing all remaining rules to cancel their transformation. 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 cancellation of all these rules and outputs the packet received through its AFailOut port. For example, in the pattern in Fig. 10, the Selector will select at most one of the four Pacman movements, non-deterministically. Visually, a Selector is depicted by a dashed-line rectangle frame with different inports and one more outport. The top left thick-lined triangle is the SuccessIn port. The top right filled triangle is the FailIn port. The top middle triangle pointing up with a line through it is the CancelOut port.

Since the semantics of DEVS is the parallel composition of atomic blocks, MoTif allows rules to be applied in parallel. This leads to what we will call "threads" of rule applications. A Synchronizer is needed to merge and synchronize 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 [16]. This avoids the undecidable problem of merging transformed models and is more memory efficient, but parallel execution requires special care. The Synchronizer waits until all the threads have sent their packets through its ASuccessIn

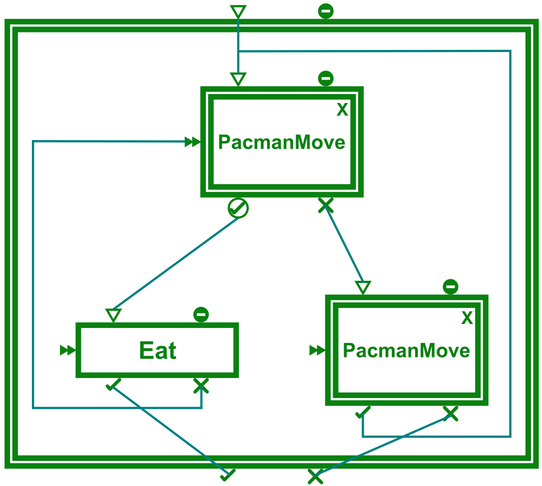


Fig. 9 MoTif model showing a recursive transformation. The PacmanMove CRule encapsulates the sub-pattern in Fig. 10. The *upper one* encodes PacMan *trying* to move and the *lower one* is to actually *make* the move. The "X" on a CRule is only for visual syntax, indicating that it contains XRules encapsulating back-tracking. The detailed semantics of this sub-model is formally described by an execution trace in the appendix

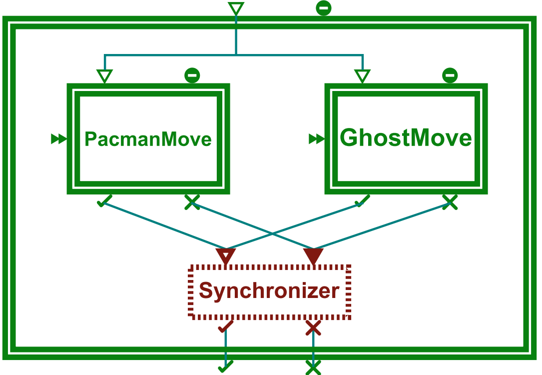


Fig. 11 An example of rules to be applied in parallel and then synchronized

or AFailIn inports. Only then will it send the transformed graph through its ASuccessOut port if at least one thread succeeded. Otherwise, it will send the unmodified graph through its AFailOut port. Figure 11 illustrates the use of the Synchronizer model. The Synchronizer (dotted line rectangle frame) synchronizes two threads: the Ghost movement and the Pacman movement.

To increase the expressiveness of the MoTif transformation language, additional rule blocks have been added. Among them is the FRule ("For all Rule") which will be used in our PacMan example. It is an ARule that applies its transformation phase to *all* the matches found (in arbitrary, but deterministic and repeatable order, through the use of a pseudo-random uniform generator) before sending the new graph. The matches are assumed to be parallel independent [10]. Two matches are parallel independent if no overlapping matched element is modified (node deletion or attribute modification) by the rule when applied. The purpose of the FRule is purely syntactic. It is syntactic sugar for applying the rule iteratively over all matched subgraphs. An FRule is represented using the same concrete visual syntax as an ARule, annotated by an "F".

The SRule ("Star Rule") is a special rule block which allows a rule to execute *as long as* possible. That is, after the received graph is matched and transformed, the resulting packet is then matched by the same rule. This continues until no more matches can be found in the resulting packet. Care should be taken when using this construct as it may result in an infinite loop. The SRule is represented using the same concrete visual syntax as an ARule, annotated by an "*".

Another special rule block is the XRule ("Transactional Rule"). It extends the ARule with memory capacity, which provides back-tracking. It keeps a stack of host graphs before applying the rule in the XRuleState. Through XRules, MoTif allows for recursion. This will be illustrated in Sect. 4.2 when analysing different types of players. The XRule is represented using the same concrete visual syntax as an ARule, annotated with an "X".

The formal semantics of each of the above blocks is detailed in Appendix A.

3.3 Soundness of MoTif

When connecting MoTif elements, by providing channels between ports, the resulting model defines an execution flow

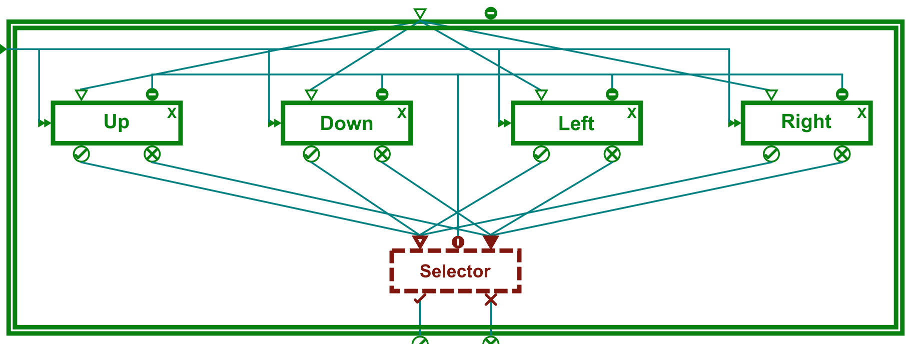


Fig. 10 The PacmanMove CRule with non-deterministic selection of XRules

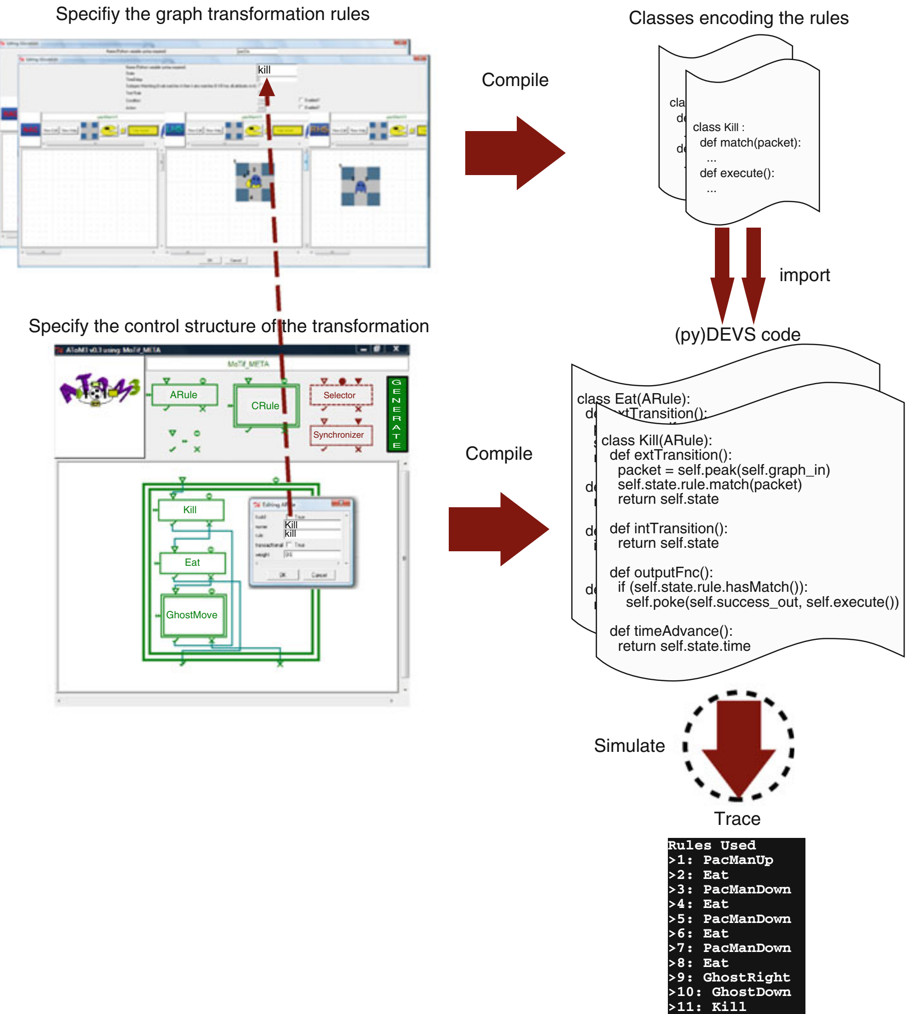


Fig. 12 DEVS-based programmed graph rewriting architecture

of a graph transformation. In order to ensure a proper flow, Proposition 1 states that whenever a packet is received by a MoTif element, a packet is eventually output from that entity.

Proposition 1 *Given an atomic MoTif element* $\langle X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \tau\rangle$*, let* $S_e \subseteq S$ *be the set of all possible states resulting from the application of* δ_{ext}*. Let* $S_f \subseteq S$ *be the set of all possible states where* τ *is finite. We then have* $\forall s \in S$, $\lambda(s)$ *is defined* $\Leftrightarrow s \in S_e \cap S_f$.

Proof The sufficient condition can be easily checked from the definition of each element. That is, if s is a state resulting from δ_{ext} and $\tau(s)$ is finite, then $\lambda(s)$ is defined. For the necessary condition, note that neither the initial state nor a state produced by δ_{int} leads to an output as the time advance is infinite. □

The MoTif language is not restricted to these constructs: pure atomic and coupled DEVS models are also allowed. The modeller can thus increase the expressiveness of the transformation model by adding customized behaviour. However, soundness of the transformation cannot be ensured if an arbitrary atomic DEVS model is involved in the transformation. By default, the MoTif code generator makes use of this feature when compiling the model down to an executable model transformation environment, by inserting a model of the interaction between the user of the transformation and the transformation model.[6] Using this generative approach, different transformation environments can easily be synthesized (allowing for example step-by-step transformation execution for debugging purposes, as opposed to interruptible run-to-completion execution).

4 Modelling the PacMan case study

At the heart of our approach lies the embedding of graphs in DEVS events and of individual transformation rules into atomic DEVS blocks. Figure 12 shows the workflow of our

[6] Using atomic DEVS models with arbitrary state and transition functions in MoTif models does not guarantee proper transformation models output as stated by Proposition 1. They can, however, be used if properly defined, as in the case study in Sect. 4 for example.

approach and the compilations it comprises. First, we model a collection of transformation rules using a domain-specific visual notation (shown on the top left of the figure). Each of these transformations is translated to a class with the same name as the rule (**`kill`** is shown here on the top right). The core of the generated code are the methods `match` which performs the matching, taking a (host) graph as argument, and `execute` which encodes the transformation phase. Second, we build a hierarchical MoTif model, in the MoTif visual modelling language (shown at the bottom left of the figure). ARules contain a reference to the appropriate rule. Third, the MoTif model gets compiled into a DEVS model: CRules get translated into coupled DEVS models and ARules models get translated into atomic DEVS models. In the latter, the `match` method encoding the matching is called in the external transition and the `execute` method encoding the transformation is called in the output function of the atomic DEVS model. The transition function is triggered by an external event (in which a to-be-transformed host graph is embedded). Note that this is a reference to a graph as in-place transformation is used. Finally, all generated code is combined and used as input to a DEVS simulator which performs the transformation and produces an execution trace.

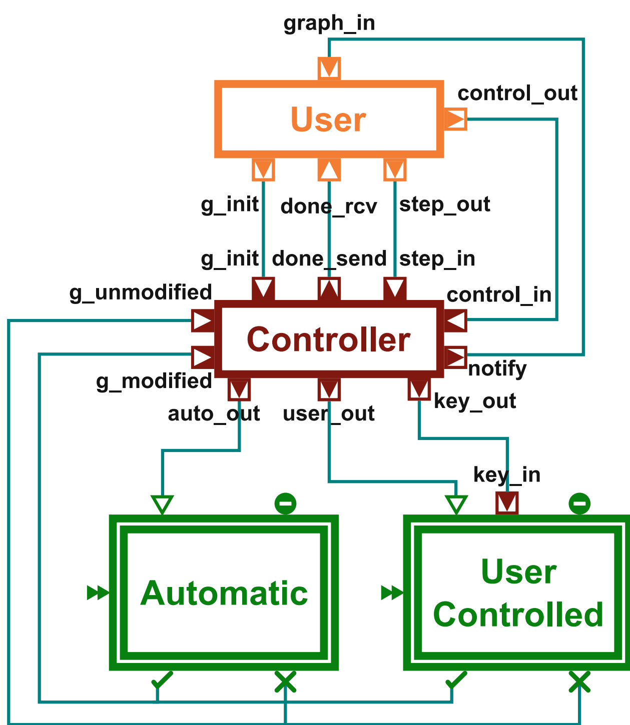


Fig. 13 The overall transformation model

4.1 (Modelling) The transformation environment

The overall transformation model of the PacMan game is shown in Fig. 13. The atomic DEVS block User is responsible for user (player) interventions. It can send the initial graph to be transformed, the number of rewriting steps to be performed (possibly infinite), and some control information. In our previous work [17], the control information was in the form of key press codes to model the user input to a game. All these events are received by the Controller, another atomic DEVS block. This block encapsulates the coordination logic between the external input and the transformation model. It sends the host graph through its outport to a rule set (the Automatic CRule) until the desired number of steps is reached. If a control event is received, however, the Controller sends the graph to another rule set (the UserControlled CRule). The Automatic CRule expects only a graph to perform the rewriting on, whereas the UserControlled CRule waits for a control, too. The details are omitted here to focus on the overall structure.

The model described in [17] does not model a realistic, playable game. When the user sends a key, the corresponding transformation rule is executed and the graph is sent to Automatic until another key is received or the PacMan entity has been deleted. What prohibits this from being suitable for a playable game is

- A rule consumes a fixed amount of time. From the graph rewriting perspective, this allows one to compute the time complexity of the transformation. However, this does not take into consideration any notion of game levels or real-time behaviour which a game such as PacMan should have.
- The user sends information to the rewriting system to (1) configure the transformation engine and (2) to control the transformation execution abstracted to the specific domain of interest (PacMan movements). This model does not take into account any playability issues, such as the Ghost moving too fast versus a user reacting too slowly.

In the sequel we present an extended model with focus on timing information. This will allow us, through simulation, to construct an optimally "playable" game.

4.2 Modelling the *player*

In current graph transformation tools, the *interaction* between the user—the player in the current context—and the transformation engine is hard-coded rather than explicitly modelled. Examples of typical interaction events are requests to step through a transformation, run to completion, interrupt an ongoing transformation, or change parameters of the transformation. In the context of the PacMan game, typical examples are game-events such as PacMan move commands. Also, if animation of a transformation is supported, the time-delay

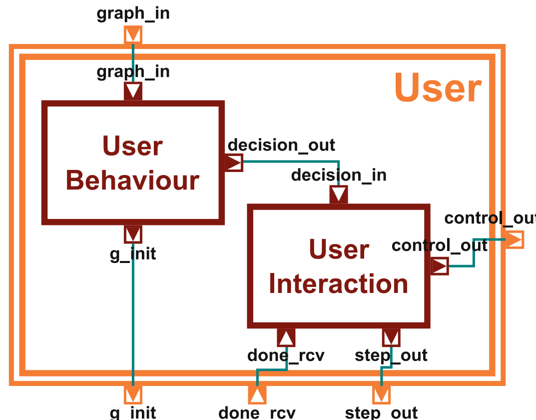


Fig. 14 The enhanced User model

between the display of subsequent transformation steps is encoded in the rewriting engine.

In contrast, in our DEVS-based approach, the interaction between the user and the game is explicitly modelled and encapsulated in the atomic DEVS block User (see Fig. 13). Note that in this interaction model, time is explicitly present.

With the current setup, it is impossible to evaluate the *quality* (playability) of a particular game dynamics model without actually *interactively* playing the game. This is time-consuming and reproducibility of experiments is hard to achieve. To support automatic evaluation of playability, possibly for *different types* of players/users, it is desirable to explicitly model player behaviour. With such a model, a complete game between a modelled player and a modelled PacMan game—an experiment—can be run *autonomously*. Varying either player parameters (modelling different types of users) or PacMan game parameters (modelling for example different intelligence levels or speed in the behaviour of Ghosts) becomes straightforward and alternatives can easily be compared with respect to playability. For the purpose of the PacMan game, player behaviour parameters can be user reaction speed or levels of decision analysis (such as path finding). We have explored these two dimensions of behaviour. Section 5 will discuss reaction speeds more in-depth.

Obviously, evaluating quality (playability) will require a precise definition of a playability *performance metric*. Also, necessary data to calculate performance metrics need to be automatically collected during experiments. Explicitly modelling player behaviour can be done without modifying the overall model described previously thanks to the modularity of DEVS. We simply need to replace the User block by a coupled DEVS block with the same ports as shown in Fig. 14. We would like to cleanly separate the way a player interrupts autonomous game dynamics, (i.e., Ghost moving) on the one hand and the player's decision making on the other hand. To make this separation clear, we refine the User block into two sub-models: the User Interaction and the UserBehaviour atomic DEVS blocks. On the one hand, the User Interaction model is responsible for sending control information such as the number of transformation steps to perform next, or a direction key to move the PacMan. On the other hand, the UserBehaviour block models the actual behaviour of the player. This is often referred to as the "AI" of a Non Player Character (NPC) in the game community. It is this block which, after every transformation step, receives the new game state graph, analyses it, and outputs a decision determining what the next game action (such as PacMan move up) will be. Also, since it is the UserInteraction block which keeps receiving the game state graph, we chose to give this block the responsibility of sending the initial host graph to the transformation subsystem.

An atomic DEVS block, Dispatch, receives the user action and branches the execution to the corresponding rule to be triggered. Thus, in this approach, the event-driven execution of the transformation is embedded in the *transformation model* rather than in the rule itself such as in [18,19]. There, the notion of Event-driven Graph Rewriting was proposed in the context of visual modelling: a graph rewriting rule is triggered in response to a user action. More precisely, the rule itself is augmented to behave according to the event it received. In MoTif, we have separated the event reaction from the rule itself. When the UserBehaviour coupled DEVS block emits the event, it is fed to the UserControlled model via the Controller as shown in detail in Fig. 15. Also, in our approach, the user and user interaction itself have been modelled in the User coupled DEVS block. Different players may use different *strategies*. Each strategy leads to a different model in the UserBehaviour block. We have modelled three types of players for our experiments: Random, Lazy, and Smart.

The Random player does not take the current game state graph into consideration but rather chooses the direction in which the PacMan will move in randomly. Note that this type of player may send direction keys requesting illegal PacMan moves such as crossing a boundary (wall). This is taken care of by our PacMan behaviour rules: the particular rule that gets triggered by that key will not find a match in the graph and hence PacMan will not move. However, time is progressing and if PacMan does not move, the ghost will get closer to it which will eventually lead to PacMan death. Note that the rules used are similar to the move right of Fig. 5: the NAC prevents movements towards a grid node already occupied by a Ghost.

The Lazy user does not make such mistakes. After querying the game state graph for the PacMan position, it moves to the adjacent grid node that has Food but not a Ghost on it. If no such adjacent grid node can be found, it randomly chooses a legal direction.

The Smart user is an improved version of the Lazy user. Whereas the Lazy user is restricted to making decisions based

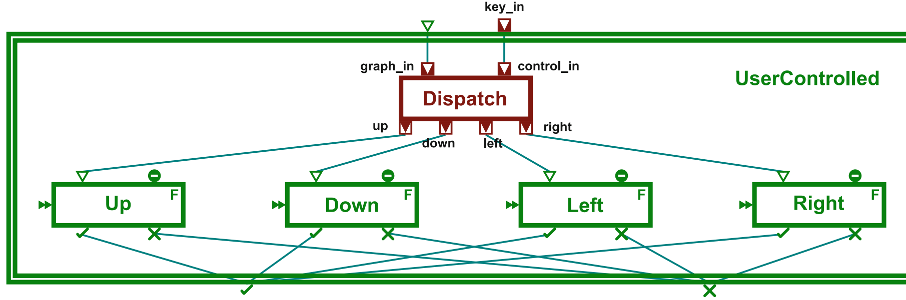


Fig. 15 The UserControlled model

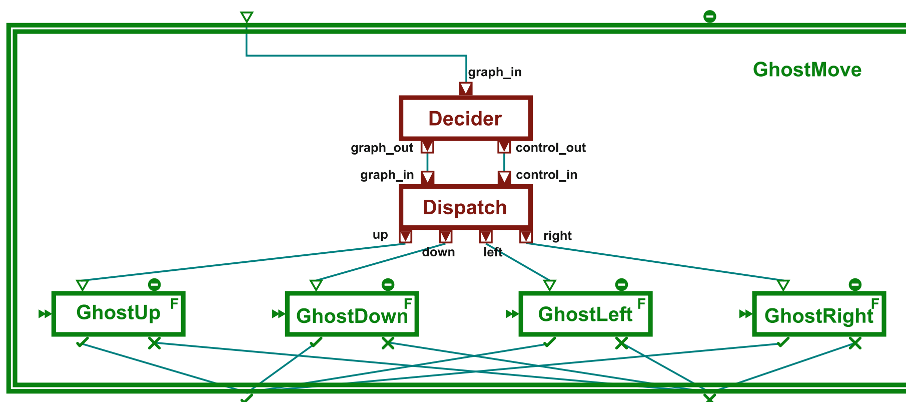


Fig. 16 The enhanced GhostMove Model

only on adjacent grid nodes, the Smart user has a "global" view of the board. The strategy is to compute the closest grid node with Food on it and move the PacMan towards it depending on the position of the Ghost. One way to implement this strategy is by using a path finding algorithm. Many solutions exist for such problems, including some efficient ones such as A* [20]. It is possible to integrate this kind of path finding techniques in a MoTif model. Because it is not the main focus of this paper, we only outline two possibilities. One is to explicitly model the path finding with transformation entities. The depth-first search model discussed in Sect. A can be a starting point. Another abstraction is algorithmic, using an atomic DEVS block to encode the algorithm in its external transition function. In any case, whether modelled declaratively using backtracking rules or algorithmically, the path finding sub-model can be used as a "black-box" and integrated transparently in the transformation model. In this case-study, extending the **PacMan** meta-model with (x, y) coordinates on grid nodes allows a linear time solution to this particular path finding problem. AToM^3 allows one to add actions to meta-model elements. Relative coordinates' management is handled in the action of each of the four associations between GridNode objects: if a GridNode instance g_1 is associated with another instance g_2 by a Left association, then $g_1.x < g_2.x$ and $g_1.y = g_2.y$. Similar conditions are defined for Right, Top, and Bottom associations. Therefore, the path finding algorithm only needs to compute the shortest Manhattan distance from PacMan to Pellet as well as perform a simple check for the grid node coordinates of the Ghost.

We compare the performance of different user behaviour types in Sect. 5. Note that to match different user types, we need to model similar strategies for the Ghost to make the game fair. Indeed, a Smart user (controlling the PacMan) playing against a randomly moving Ghost will not be interesting nor will a Lazy user playing against a Smart Ghost. As players may become better at a game over time, game *levels* are introduced whereby the game adapts to the player's aptitude. This obviously increases game playability.

4.3 Modelling the *game*

As long as the (modelled) player does not send a decision key to move the PacMan (thus changing the game state graph) the game state graph continues to loop between the Controller block and the Automatic block. If no instantaneous rule (Kill or Eat) matches, then it is the lower priority GhostMove block that modifies the graph. The Ghost movement model in Fig. 16 is enhanced from earlier work [17] to support different strategies. The game state graph is received by a Decider atomic DEVS block. Similar to the UserBehaviour block, it emits a direction that drives the movement of the Ghost. The Random, Lazy, and Smart strategies are analogous to those of the player. The Random Ghost

will randomly choose a direction, the Lazy Ghost will look for a PacMan among the grid nodes adjacent to the one the Ghost is on, and the Smart Ghost has "global" vision, and always decides to move towards the PacMan.[7] The same argument previously made about optimal path finding and backtracking applies. The Decider sends the graph and the decision (in the form of a key) to a Dispatch block and the rest of the behaviour is identical to that in the UserControlled CRule.

4.4 Explicit use of time

We have now modelled both game and player, and the behaviour of both can use Random, Lazy, or Smart strategies. However, one crucial aspect has been omitted up to now: the notion of time. Time is critical for this case study since game playability depends heavily on the relative speed of player (controlling the PacMan) and game (Ghost). The speed is determined by both decision (thinking) and reaction (observation and keypress) times.

We will now show how the notion of time from the DEVS formalism integrated in a graph transformation system can be used for realistic modelling of both player and game. We consider a game to be unplayable if the user consistently either wins or loses. The main parameter we have control over during the design of a PacMan game is the speed of the Ghost.

Each atomic DEVS block has a state-dependent time advance that determines how long the block stays in a particular state. Kill and Eat rules should happen instantaneously, thus their time advance is 0 whenever they receive a graph. In fact, all rules involved in the transformation have time advance 0. What consumes time is the decision making of both the player (deciding where to move the PacMan) and the game (deciding where to move the Ghost). For this reason, only the Decider and the UserBehaviour blocks have strictly positive time advance.

To provide a consistent playing experience, the time for the Ghost to make a decision should remain almost identical across multiple game plays. The player's decision time may vary from one game to another and even within the same game. We have chosen a time advance for the Decider that is sampled from a uniform distribution with a small variance (interval radius of 5 ms). What remains is to determine a reasonable average of the distribution. To make the game playable, this average should not differ significantly from the player's reaction time. If they are too far apart, a player will consistently lose or win making the game uninteresting.

[7] In the original PacMan video game, these different Ghost types are referred to as "Clyde" for Random, "Pinky" for Lazy, and "Inky" for Smart.

5 Simulation experiments

In the previous Section, we determined that the playability of the PacMan game depends on the right choice of the average time advance of the Decider block, i.e., the response time of the Ghost. We will now perform multiple simulation experiments, each with a different average time advance of the Decider block. For each of the experiments, a playability performance metric (based on the outcome and duration of a game) will be calculated. The value of the Decider block's average time advance which maximizes this playability performance metric will be the one retained for game deployment. Obviously, the optimal results will depend on the type of player.

5.1 Modelling user reaction time

First of all, a model for player reaction time is needed. Different psychophysiology controlled experiments [21] give human reaction times (for subjects between the ages of 17 and 20):

- the time of simple visuomotor reaction induced by the presentation of various geometrical figures on a monitor screen with a dark background;
- the time of reaction induced by the onset of movement of a white point along one of eight directions on a monitor screen with a dark background.

The reaction time distribution can be described by an asymmetric normal-like distribution. The cumulative distribution function for sensorimotor human reaction time is

$$F(x) = e^{-e^{\frac{b-x}{a}}}$$

where a characterizes data scatter relative to the attention stability of the subject: the larger a is, the more attentive the subject; b characterizes the reaction speed of the subject. For simulation purposes, sampling from such a distribution is done by using the Inverse Cumulative Method [22]: a value sampled from the initial distribution function is computed by sampling a random number from a uniform distribution in the interval [0, 1] and subsequently evaluating the inverse cumulative function at that value.

For our simulation, four types of users were tested: Slow with $a = 33.3$ and $b = 284$, Normal with $a = 19.9$ and $b = 257$, Fast with $a = 28.4$ and $b = 237$, VeryFast with $a = 17.7$ and $b = 222$. The parameters used are those of four example subjects in [21].

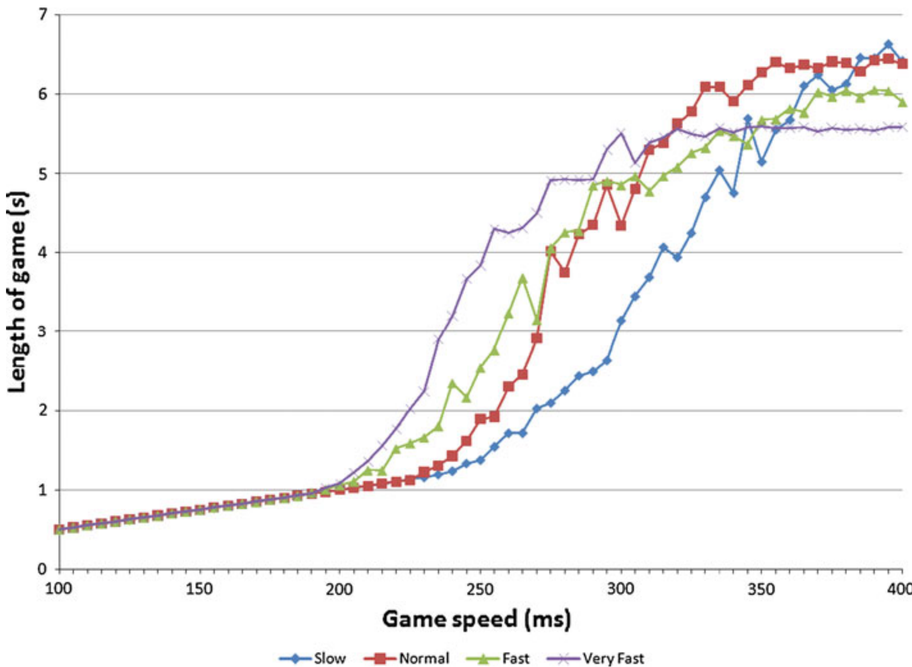


Fig. 17 Time till end

5.2 Simulation results

For the simulations, we only consider the Smart user strategy. For each type of user (Slow, Normal, Fast, and VeryFast), the *length of the simulated game* is measured: the time until PacMan is killed (loss) or no Pellet is left on the board (victory). To appreciate the need for these results, the score is also measured for each run. Simulations were run for a game configuration with 24 GridNodes, 22 Pellets, 1 Ghost, and 1 PacMan. The game speed (ghost decision time) was varied from 100 to 400 ms. Each value is the result of an average over 100 samples simulated with different seeds.

The following presents the simulation results obtained by means of the DEVS simulations of our game and player model. All figures show results for the four types of users (Slow, Normal, Fast, and VeryFast). Figure 17 shows the *time until the game ends* as a function of the time spent on the Ghost's decision. The increasing shape of the curves imply that the slower the ghost, the longer the game lasts. This is because the user has more time to move the PacMan away from the Ghost. One should note that after a certain limit (about 310 ms for the VeryFast user and 350 ms for the Normal user), the curves tend to plateau. To further investigate behaviour, Fig. 18 shows the score (relative percentage of number of Pellets eaten) for the different game speeds. Not surprisingly, these curves and the previous ones have the same shape: increasing up to a certain limit and then remaining constant. These limits even coincide at sensibly the same values and happen when the score reaches 100%. An explanation for this behaviour is simply that after a certain point, the Ghost decision time is too low and the user always wins. Therefore, the optimal average *time-advance* value we are looking for is found in the middle of the steep slope of the plots.

Figure 19 depicts the *frequency* with which a player will *win* a game (when playing a large number of games) as a function of the time spent on the Ghost's decision. We decided that we want to deploy a game where the user should be able to win with a probability of 75%. Thus, the optimal average Ghost time advance (decision time) was found to be 325 ms (taking into account fairness among the different types of users). Note that the focus of this paper is not on the reasons for such decisions/assumptions but rather on how the integration of graph rewriting systems with the DEVS formalism gives the modeller the right level of abstraction for simulation and performance analysis.

To give further insight into the variability of the game experience, Fig. 20 shows the *game length distribution* at the optimal *time advance* value. It is an unimodal distribution with a peak at 7.5 s. This average is quite low, but not surprising given the small game board. Experience with the deployed real-time game application is consistent with this value.

5.3 Game deployment

Having found a prediction for the optimal time the Decider block should spend on the choice of the next movement of the ghost entity, we can now test the simulated game with real users, in real-time. From the AToM^3

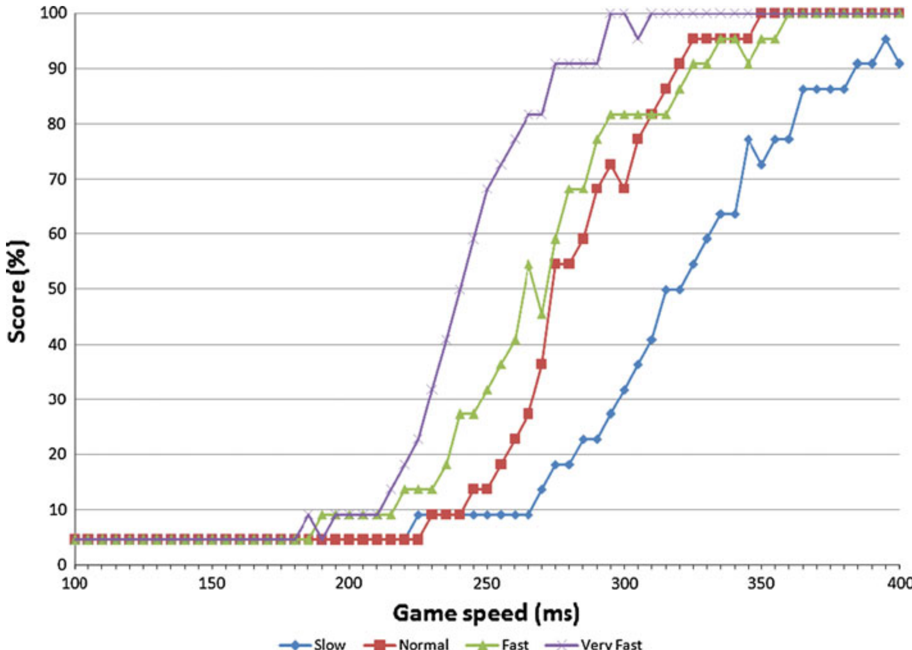


Fig. 18 Score at the end of game

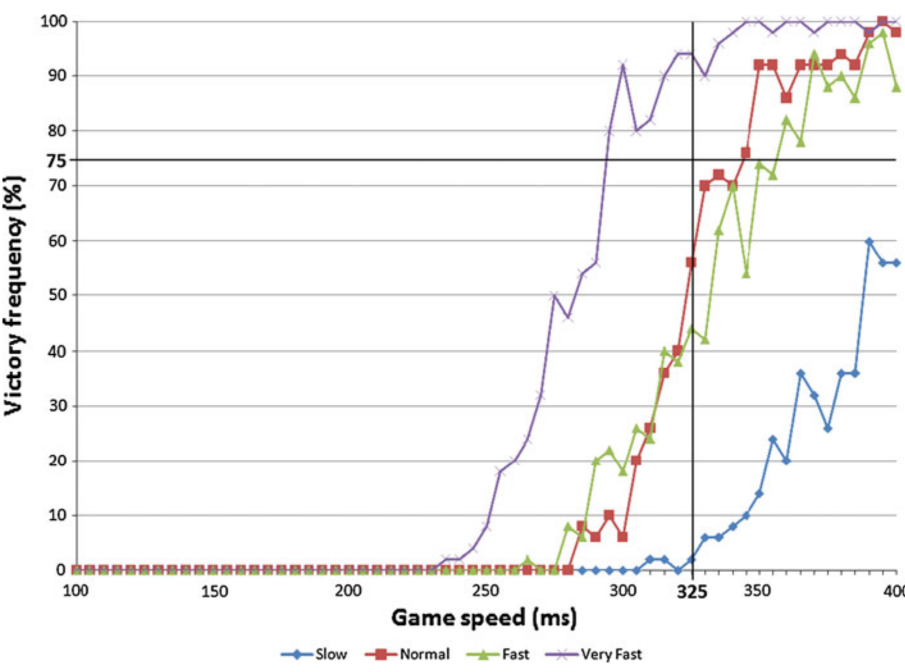


Fig. 19 Victory frequency

PacMan formalism/meta-model we have synthesized—yet another model transformation—an Ajax/SVG-based application. The MoTif model is slightly modified to get the decision event from an external source. The User coupled DEVS now has an inport *"interrupt"* waiting for external input. The User Interaction block is then linked to that port to receive the decisions. The player behaviour model in the User Behaviour block is thus discarded. The transformation model is executed by a real-time version of our `pythonDEVS` simulator. An intermediate layer for event management and communication between the web client and the transformation model is used.

The most expensive operation in a graph transformation tool is the matching phase. MoTif's current implementation of the rule matching turned out to be significantly fast enough to play the deployed game. Figure 21 shows a snapshot of the deployed game with the same initial state and conditions as the simulated game.

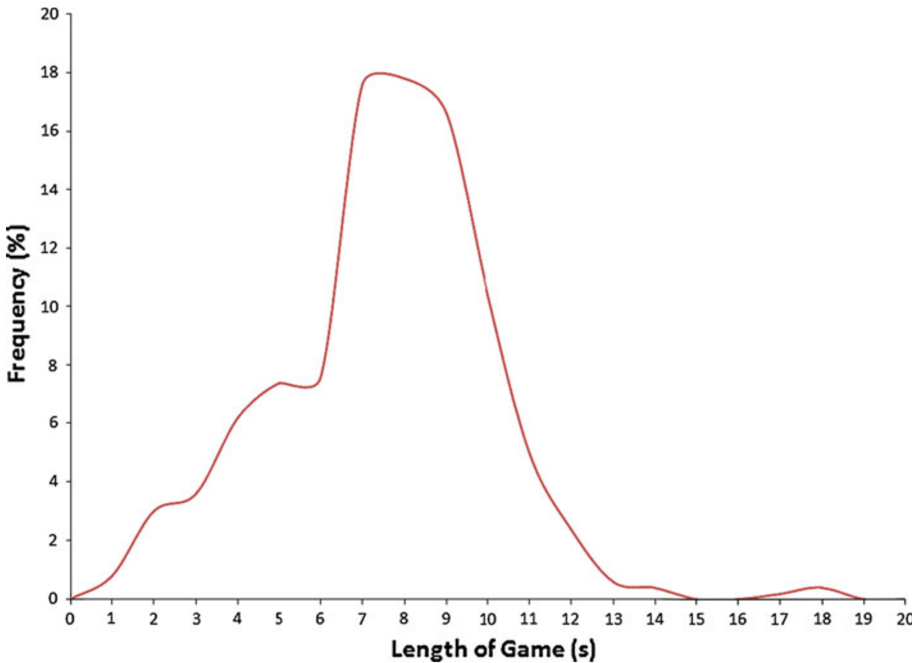


Fig. 20 Game length distribution; Normal user, game time advance 325 ms

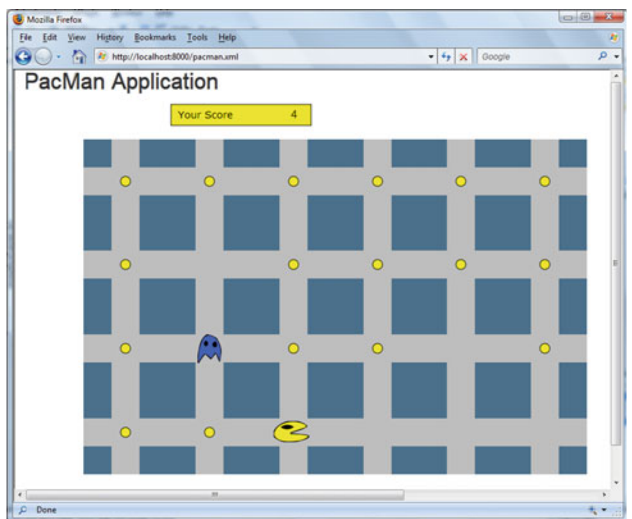


Fig. 21 Snapshot of the deployed PacMan game running in a web browser

6 Related work

6.1 Graph transformation with time

Timed Graph Transformation (TGT), as proposed by Gyapay et al. [23], 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 previously. In contrast with TGT, it is the execution of a rule that can increase time and not the rule itself. That is, in TGT the designer of the rule can increment the chronos element in the RHS pattern. However, in MoTif time is a property of a rule taken into consideration in the scheduling of the rules. Hence, the control flow (of the graph transformation) has full access to time. This addition of time is similar to how Heckel et. al. define stochastic graph transformations [24]. Every transformation rule is augmented by a probability indicating the delay of application of the rule. In fact, Heckel et al.'s approach is complementary to ours: one can encode the application rate of individual MoTif rules in the time advance function, if the current simulation time is stored in the state of each rule. As pointed out in [23], 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 retains this expressiveness.

6.2 Tool comparison

Many graph transformation tools and languages have been developed during the last two decades. Hence, we present those that describe a transformation in a controlled way,[8] (i.e., programmed graph rewriting). Table 1 compares them to MoTif using the criteria enumerated in Sect. 1.

GReAT [4,25,26] is the model transformation language for the domain-specific modelling tool GME [27]. GReAT's control structure language uses a custom asynchronous dataflow diagram notation where a production is represented by a "block" (called Expression in [4]). Expressions have input and output interfaces (*inports* and *outports*). They

[8] This is by no means an exhaustive list.

Table 1 A comparison of the control structure of graph transformation tools

Property	MoTif	ProGReS	FUJaBA	GReAT	VMTS
Control structure	DEVS	Imperative language	Story Diagram	Data flow	Activity diagram
Atomicity	ARule	`transaction`, rule	Rule	Expression	Step
Sequencing	Yes	`&`	Yes	Yes	Yes
Branching	Selector pattern	`choose...else`	Branch activity	Test/Case	Decision Step, OCL
Looping	FRule, SRule	`loop`	For-all pattern	Yes	Self loop
Non-determinism	Selector pattern	`and`,`or`	No	$1-n$ connection	No
Recursion	Yes	Yes	No	Yes	Yes
Parallelism	Synchronizer pattern	No	Optional	No	Fork, Join
Back-tracking	XRule	Implicit	No	No	No
Hierarchy	CRule	Modularisation	Nested state	Block, ForBlock	High-level Step
Time	Yes	No	No	No	No

exchange packets: node binding information. The in-place transformation of the host graph thus requires only packets to flow through the transformation execution. Upon receiving a packet, if a match is found, the (new) packet will be sent to the output interface. Inport to outport connections depict sequencing of expressions in that order.

Two types of hierarchical rules are supported. A Block forwards all the incoming packets of its inport to the target(s) of that port connection, (i.e., the first inner expression(s) of the Block). On the other hand, a ForBlock sends one packet at a time to its first inner expression(s). When the ForBlock has completely processed a packet, the next packet is sent and so on. Branching is achieved using Test expressions. Test is a special composite expression holding Case expressions internally. A Case is given in the form of a rule with only a LHS and a boolean condition on attributes. An incoming packet is tested on each Case and every time the Case succeeds, it is sent to the corresponding outport. If a Case has its *cut* behaviour enabled, the input will not be tried with the subsequent Cases. When an outport is connected to more than one inport or if multiple Cases succeed in a Test (also one-to-many connection), the order of execution of the following expressions is non-deterministic. To achieve recursion, a composite expression (Block, ForBlock, or Test/Case) can have an internal connection to a parent or ancestor expression (in terms of the hierarchy tree).

In **VMTS** [3,5], the controlled graph rewriting system is provided by the VMTS Control Flow Language (VCFL), a stereotyped UML Activity Diagram. In this abstract state-machine a transformation rule is encapsulated in an activity, called Step. Sequencing is achieved by linking steps; self loops are allowed. Branching in VCFL is a Decision Step conditioned by an OCL expression. Chains of Steps can thus be connected to the Decision. However, at most one of the branches may execute. The Steps connected to the Decision should then be non-overlapping (this is checked at compile-time). A branch can also be used to provide conditional loops and thus support iteration.

Steps can be nested in a High-level Step. A primitive step ends with success when the terminating state is reached and 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. As in GReAT, recursive calls to High-level Steps are possible. A Fork connected to a Step allows for parallelism and a Join synchronizes the parallel branches. Semantically, parallelism is possible in VMTS but it is not yet implemented [5].

The Programmed Graph Rewriting System (**PROGReS**) was the first fully implemented environment to allow programming through graph transformations [6,28,29]. The control mechanism is an imperative language (with textual concrete syntax). A rule in ProGReS has a boolean behaviour indicating whether it succeeded or not. Among the imperative control structures it provides, rules can be sequenced using the `&` operator. Branching is supported by the `choose` construct, which applies the first applicable rule following the specified order. ProGReS allows non-deterministic execution of transformation rules. `and` and `or` are the non-deterministic duals of `&` and `choose`, respectively, by selecting in a random order the rule to be applied. With the `loop` construct, it is possible to loop over sequences of (one or more) rules as long as they succeed.

A sequence of rules can be encapsulated in a `transaction` following the usual atomicity, isolation, durability, and consistency (ACID) properties. The underlying database system where the models are stored is responsible for ensuring the first three properties. An implicit back-tracking mechanism ensures consistency. Hence, ProGreS offers two kinds of back-tracking: data back-tracking (with undo operations) and control flow back-tracking [30]. When a rule r' fails in a sequence in the context of a trans-

action, the control flow will back-track to the previously applied rule r. The data back-tracking mechanism undoes the changes performed by the transformation of r. If r is applicable on another match, it applies the transformation on it and the process continues with the next rule (possibly r'). If r has no further matches, two cases arise. If r was chosen non-deterministically from a set of applicable rules, a non-previously applied rule is selected from this set. Otherwise, the process back-tracks recursively to the rule applied before r. Sequences and transactions can be named allowing for recursive calls. The module concept provides a two-level hierarchy in the control flow structure by encapsulating a sequence of transactions.

Insights gained through the development of PROGReS have led to **FUJaBA** (From UML to Java and Back Again) [2,31], 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 [31] and Statecharts. An activity in such a diagram contains either graph rewrite rules, which adopt a 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. Although the standard version of FUJaBA does not include the notion of time, the real-time [32,33] version does.

The **MOFLON** [7] toolset uses the FUJaBA engine for graph transformation, since the latter already features UML-like graphical diagrams. It provides an environment where transformations are defined by Triple Graph Grammars (TGGs) [34]. 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 the above 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 event-triggered execution, but the user and his interaction with the executing transformation can be explicitly modelled, offering a user-centric approach to model transformations. On top of the novelties MoTif adds to control structures for model transformation, it is the only language introducing the notion of time and allowing the designer to *explicitly model* back-tracking and recursion behaviours.

Note that in the aforementioned tools, user-tool interaction is hard-coded. Furthermore, the notion of time is absent in all of these languages. Some do provide sophisticated, user-friendly graphical interfaces. Efficiency and expressiveness of the rule pattern language is not the focus of this paper. A comparison with these criteria can be found in the results of graph transformation tool contests [35]. The above list of tools can be extended by many other existing tools. For example, the control structure of AGG [36] is layer-based, AToM^3 is priority-based, MoTMoT [37] relies on story diagrams, and VIATRA2 [38] relies on an abstract state machine, just to name a few others.

In [17], we showed the advantages of re-using a discrete-event modelling/simulation formalism to describe transformation control. In [39], we focused on the time aspect of modelling complex transformations, a side-effect of using a discrete-event modelling formalism. The present paper is an extension of [39]. First, formalization of the transformation language *MoTif*, was provided. Furthermore, new constructs were added to increase the expressiveness of the language, such as the Synchronizer to merge parallel threads of transformation, the FRule allowing a rule to be atomically applied on all its matches, and the XRule to add recursion and back-tracking to the language. Second, we elaborated on how the PacMan example was modelled and on the analysis of the simulations experiments. An outline of how the game was finally synthesized and deployed in a web application was also given. Finally, we have compared other graph transformation languages with MoTif.

Nowadays, the Eclipse Modelling Framework (EMF) is gaining a lot of popularity. EMF allows to design a transformation language by modelling its abstract and concrete syntaxes. MoTif is a completely modelled language. Its abstract syntax (defined by a meta-model) and its concrete (visual) syntax are specified in AToM^3. This can also be done in EMF. The semantics of MoTif is defined in terms of the DEVS formalism. The DEVS simulator ensures the execution of MoTif transformations. This would not be directly possible to implement in EMF, since this would require to add a virtual machine for simulating DEVS models on top E-Core virtual machine.

For a more elaborate comparison, see the Ph.D. thesis [40].

6.3 Other approaches

A possible approach to ensure the incremental synchronization of models is incremental transformation. Typically, if a change occurred in a model, the effects of these changes should be propagated to other related models, preferably without re-executing the entire model transformation from the beginning. An incremental transformation is defined as a set of relations between source and target meta-models. These relations define constraints on models to be

synchronized. Change-detection and change propagation mechanisms are then used as in [41]. The first time it is run, the transformation creates a target model. In Tefkat [42], trace links are automatically created. Then, if a change is detected in one of the two models, it propagates this change to the other model, by adding, removing, or updating elements so that the relations are still satisfied. Ráth et. al. [41] propose a technique based on the RETE algorithm that consists of caching the matches of a pattern for future rule applications. The match set is thus available from the cache at any time without having to perform further pattern matching. The cache is incrementally updated whenever changes are made to the model.

MoTif is designed by combining graph transformation primitives (rules) with DEVS as a scheduling language. Similarly, Fujaba's story diagrams combine rules with activity diagrams as a scheduling language. More recently, Feng in his Ph.D. [43] thesis combines rules with Ptolemy's synchronous data flow. An interesting avenue for future research is a combination of rules with languages such as MATLAB Simulink [44] or SCADE [45].

7 Conclusion

In this paper, we have introduced MoTif as a transformation language based on the Discrete-EVent system Specification (DEVS) formalism for the specification of complex control structures for programmed graph rewriting, with time. DEVS allows for highly modular, hierarchical modelling of timed, reactive systems. In our approach, graphs are embedded in events and individual rewrite rules are embedded in atomic DEVS models. The metamodel of MoTif was described together with a formalization of the different entities of the language. A side effect of this approach is the introduction of an explicit notion of time. This allows one to model a time-advance for every rule as well as to interrupt (pre-empt) rule execution. The approach we described elegantly satisfies all the requirements for a programmed model transformation language enumerated at the beginning of this paper.

We have shown how the explicit notion of time allows for the simulation-based design of reactive systems such as modern computer games. We used the well-known game of PacMan as an example and modelled its dynamics with programmed graph transformation based on DEVS. This allowed the modelling of player behaviour, incorporating data about human players' behaviour and reaction times. We used the models of both player and game to evaluate, through simulation, the playability of a game design. In particular, we proposed a playability performance metric and varied parameters of the PacMan game. This led to an "optimal" (from a playability point of view) game configuration. The user model was subsequently replaced by a web-based visual interface to a real player, and the game model was executed using a real-time DEVS simulator.

The use of graph transformation at the heart of this approach allows non-software-experts to specify all aspects of the design in an intuitive fashion. The resulting simulations give quantitative insight into optimal parameter choices. This is an example of Modelling and Simulation Based Design, where the graph transformation rules and the timed transformation system are modelled, as well as the user (player) and the context. Having modelled all these aspects in the same model transformation framework, *MoTif*, allows for simulation-based design.

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). 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 send graph through their ports. Other ports are used to 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 [46] formalism can be used.

Using the DEVS formalism as a control flow language for graph rewriting enabled us not only to model the AToM^3 semantics for graph transformation execution but also to model 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.

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 rule-based graph transformation systems.

For future work we propose the following. Currently, back-tracking in XRules is done by making local copies of the graph. This is obviously hard to maintain in memory as graphs get larger, (e.g., order of magnitude 10^6 number of nodes). One possibility is to exploit a transaction mechanism for XRules. Although the transformation implementation is fast enough for this specific example of a PacMan game, performance analysis is needed for larger-scale games.

The synchronization process of parallel transformations in MoTif assumes parallel independence of the rules and thread-safe interaction between the parallel transformation threads. We will to investigate what restrictions on MoTif transformations are necessary to satisfy these assumptions.

At the model structure level, it is noted how topologically similar the UserControlled rules and GhostMove CRules are. Re-use and parametrization of transformation models deserves further investigation.

For the presented Modelling and Simulation-based design application, we could also enhance the game with Dynamic Difficulty Adjustment techniques as outlined in [47]. For example, the user speed could be measured in real-time and compared with the simulated user speeds. The speed of the ghost can then be adapted appropriately.

Appendix A: Detailed semantics of MoTif

In the following, we formalize each of the MoTif blocks in terms of DEVS structures. The time base used is $T = \mathbb{R}^+ \cup \{+\infty\}$. Also, for the sake of completeness of the formal DEVS models, we assume an input segment function[9] $\omega : T \rightarrow X$ which determines the input event on an inport at a certain time.

A.1 The ARule

The ARule is an atomic DEVS, parametrized by a rule r and two boolean parameters σ_1 and σ_2 to indicate whether a pivot is to be sent upon a successful or failed matching, respectively. The parameter Δ specifies the time a rule will consume during the matching phase. An ARule is defined by the following structure:

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

The state S is defined as

$$S = \Big\{ (\gamma, \rho, \alpha, \sigma_1, \sigma_2, \Sigma(r)) \,|\, \gamma \in G^*, \rho = m(\gamma), \alpha, \sigma_1, \sigma_2 \in \{\texttt{true}, \texttt{false}\} \Big\}.$$

9 ω triggers the external transition.

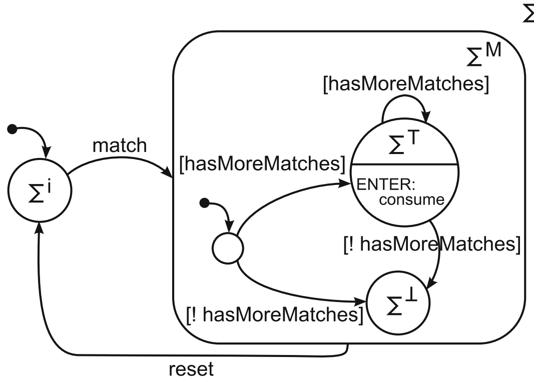


Fig. 22 The state automaton Σ underlying the execution of a rule

In this notation, γ is a graph (model) taken from the set G of all directed, labelled, attributed, typed graphs.[10] ρ is a single-node graph such that $\rho \overset{m}{\rightarrow} \gamma$ is a morphism. For simplicity, ρ will from now on refer to the single node representing the optional pivot. The boolean variable α indicates whether the ARule is active. $\Sigma(r)$ is a hierarchical state machine with guards and events, indicating the state of the rule r. The automaton in Fig. 22 shows the different modes a rule can be in and will be explained when defining δ_{ext}. S is initially set to $s_0 = (\texttt{nil}, \texttt{nil}, \texttt{false}, \sigma_1, \sigma_2, \Sigma^i(r))$.

The time-advance is finite only when the ARule is active:

$$\tau(s) = \begin{cases} \Delta \in [0, +\infty) & \text{if } \alpha = \texttt{true} \\ +\infty & \text{otherwise} \end{cases}, \quad \forall s \in S.$$

The AModelIn port can receive a packet $\langle \gamma, \rho \rangle$ consisting of a graph γ and possibly a pivot ρ. Therefore, $X_{\texttt{AModelIn}} = \{\langle \gamma, \rho \rangle\} \cup \{\phi\}$. ϕ represents the null event as used in [11]: it covers the case when no event is present. The ANextIn port can receive any event instance E of a sub-class of Event. Hence, $X_{\texttt{ANextIn}} = \texttt{INSTANCEOF}(\text{Event}) \cup \{\phi\}$. The ACancelIn port can only receive $\bot$, indicating to the rule not to apply its transformation phase. Thus $X_{\texttt{ACancelIn}} = \{\bot\} \cup \{\phi\}$. We can then define the input set of an ARule as the cross-product: $X = X_{\texttt{AModelIn}} \times X_{\texttt{ACancelIn}} \times X_{\texttt{ANextIn}}$.

The output set of an ARule is

$$Y = Y_{\texttt{ASuccessOut}} \times Y_{\texttt{AFailOut}},$$
$$Y_{\texttt{ASuccessOut}} = \{\langle \gamma', \rho' \rangle\} \cup \{\phi\},$$
$$Y_{\texttt{AFailOut}} = \{\langle \gamma, \rho \rangle\} \cup \{\phi\}.$$

$\langle \gamma', \rho' \rangle$ is the resulting packet after the transformation phase of the application of rule r. Note how it is possible to have $\rho' = \rho$.

The internal transition function passivates the ARule:

$$\delta_{\text{int}}(\gamma, \rho, \alpha, \sigma_1, \sigma_2, \Sigma(r)) = (\gamma, \rho, \texttt{false}, \sigma_1, \sigma_2, \Sigma(r)).$$

10 $G^* = G \cup \{\texttt{nil}\}$ and $V^*_\gamma = V_\gamma \cup \{\texttt{nil}\}$.

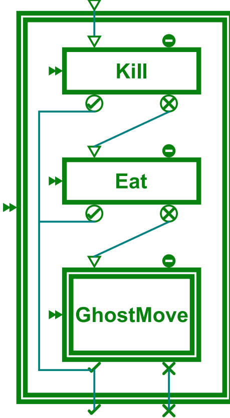


Fig. 23 The Automatic CRule shows sequencing of FRules and a CRule

The external transition function is constructed as follows:

$$\delta_{\text{ext}}((s,e),x) = \begin{cases} (\gamma, \rho, \texttt{false}, \sigma_1, \sigma_2, \Sigma(r)) & \text{if } x = \perp \\ (\gamma, \rho, \texttt{true}, \sigma_1, \sigma_2, \Sigma^M(r)) & \text{if } x = \langle \gamma, \rho \rangle \vee x = E \end{cases}.$$

When $\perp$ is received (from ACancelIn), the ARule is deactivated but the state of r is preserved. When a packet is received (from AModelIn), first the state of r is reset. This clears previous matches and dereferences any pointer assigned to a previous match. Then, r enters in match mode. This is denoted by Σ^M because, according to the statechart in Fig. 22, r can be in either Σ^T if there is at least one unprocessed match left or $\Sigma^{\perp}$ if not. In case an event is received from ANextIn (this is when $x = E$), r selects the next match, if possible. Again, r results in either Σ^T or $\Sigma^{\perp}$ states.

Finally, the output function returns the transformed packet if the match was successful; otherwise, it returns the original packet. INSTATE checks whether the automaton Σ is in a particular state (basic or composite).

$$\lambda(s) = \begin{cases} \langle \gamma', \rho' \rangle & \text{if } \sigma_1 = \texttt{true} \wedge \Sigma(r).\texttt{INSTATE}\left(\Sigma^{\top}(r)\right) \\ \langle \gamma', \texttt{nil} \rangle & \text{if } \sigma_1 = \texttt{false} \wedge \Sigma(r).\texttt{INSTATE}\left(\Sigma^{\top}(r)\right) \\ \langle \gamma, \rho \rangle & \text{if } \sigma_2 = \texttt{true} \wedge \Sigma(r).\texttt{INSTATE}\left(\Sigma^{\perp}(r)\right) \\ \langle \gamma, \texttt{nil} \rangle & \text{if } \sigma_2 = \texttt{false} \wedge \Sigma(r).\texttt{INSTATE}\left(\Sigma^{\perp}(r)\right) \end{cases}.$$

Let us consider the Kill ARule in Fig. 23. Let $r = \textbf{kill}$ (the rule in Fig. 2) and $\Delta = 0$. **kill** is the rule encoded in the state of this ARule. Note that $\sigma_1 = \sigma_2 = \texttt{true}$ (sends a pivot in case of both a failed or a successful match). Kill's state is initially s_0 as defined previously. At a certain time, a packet $\langle \gamma, \rho \rangle$ is received. After resetting previous matches, **kill** matches the host graph against its LHS. Assuming a match is found, **kill** is in its $\Sigma^{\top}$ state. Since $\Delta = 0$, the output function occurs immediately: the transformation is applied and the new packet $\langle \gamma', \rho' \rangle$ is sent through ASuccessOut. Finally, the internal transition function is triggered and **Kill** is deactivated until an event is received from AModelIn or ANextIn, since the current time-advance evaluates to $+\infty$. In the case where no matches were found, **kill** remains in its present state. As a result, the output function sends the host packet through AFailOut.

A.2 The CRule

The CRule is defined exactly like a coupled DEVS:

$$CRule = \Big\langle X, Y, N, M = \{M_i | i \in N\},$$
$$I = \{I_i\}, Z = \left\{Z_{i,j}\right\}, \texttt{select}\Big\rangle.$$

The input and output sets are defined as in their atomic counterpart:

$$X = X_{\texttt{CModelIn}} \times X_{\texttt{CCancelIn}} \times X_{\texttt{CNextIn}},$$
$$Y = Y_{\texttt{CSuccessOut}} \times Y_{\texttt{CFailOut}}.$$

For example, the CRule Automatic in Fig. 23 has three inner models: two ARules Kill and Eat and one CRule GhostMove. M is the set of these three inner models (in general all ARules, CRules, or other DEVS sub-models of this CRule). N is the set of labels which identifies each component by its unique name (as it appears in Fig. 23 for example). The connection topology is given by $I_{\text{Kill}} = \{\text{Automatic}\}, I_{\text{Eat}} = \{\text{Kill}\}, I_{\text{GhostMove}} = \{\text{Eat}\}$, and $I_{\text{Automatic}} = \{\text{Kill, Eat, GhostMove}\}$. The $Z_{i,j}$ functions are all the identity, as in the example of Sect. 3.1. In this ElemCRule, first the **kill** rule is tried. If it fails, then **eat** is tried and, in case of failure, **ghostMove** is tried. If any of the rules in this CRule matches, the resulting transformed packet is sent out of Automatic. This encodes priorities in a transformation.

As for the select function, it chooses one sub-model of the CRule from the *imminent set*. The imminent set is the set of sub-models from M which would have an internal transition at the same time. This set is computed at simulation time by the simulator, as described in [11]. The select function is described by the following prioritized algorithmic steps:

1. If a Selector is in the imminent set, choose the Selector: this ensures that only one rule will execute.
2. Among all the rules that still have a match, choose a corresponding ARule from the imminent set at random: this emulates to non-deterministic execution of a rule.
3. At this point no rule has any unprocessed matches left. Now choose any of the ARule models in the imminent set.
4. Finally, the imminent set contains either custom atomic blocks or Synchronizers. Select a model randomly.

Once a sub-model is selected, it first produces an output, if needed for the current state. This may trigger the δ_{ext} of the influences of this sub-model. Then its δ_{int} is performed.

The `select` function is called as long as the imminent set is non-empty.

A.3 The Selector

The Selector is also an atomic DEVS:

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

The state S of the Selector is composed of two boolean variables t and f, representing the reception of any event from ASuccessIn and AFailIn, respectively. The event is stored in π.

$$S = \left\{ (t, f, \pi) \,|\, t, f \in \{\texttt{true, false}\}, \pi \in \texttt{INSTANCEOF}\,(\text{Event}) \cup \{\emptyset\} \right\}.$$

S is initially set to $s_0 = (\texttt{false}, \texttt{false}, \emptyset)$.

As for the ports, the event class Event' is the same as Event. We use the two notations to distinguish an event E' received by ASuccessIn from E received by AFailIn.

$$\begin{aligned}
X &= X_{\texttt{ASuccessIn}} \times X_{\texttt{AFailIn}},\\
X_{\texttt{ASuccessIn}} &= \{\text{Event}'\} \cup \{\phi\},\\
X_{\texttt{AFailIn}} &= \{\text{Event}\} \cup \{\phi\},\\
Y &= Y_{\texttt{ASuccessOut}} \times Y_{\texttt{AFailOut}} \times Y_{\texttt{ACancelOut}},\\
Y_{\texttt{ASuccessOut}} &= \{\text{Event}'\} \cup \{\phi\},\\
Y_{\texttt{AFailOut}} &= \{\text{Event}\} \cup \{\phi\},\\
Y_{\texttt{ACancelOut}} &= \{\bot\} \cup \{\phi\}.
\end{aligned}$$

The time-advance returns 0 if an event was received:

$$\tau(s) = \begin{cases} 0 & \text{if } t = \texttt{true} \vee f = \texttt{true} \\ +\infty & \text{otherwise} \end{cases}, \forall s \in S.$$

The external transition function follows from the above definitions:

$$\delta_{\text{ext}}((s, e), x) = \begin{cases} (\texttt{true}, \texttt{false}, E') & \text{if } x = E' \\ (\texttt{false}, \texttt{true}, E) & \text{if } x = E \end{cases}.$$

The output function sends two events simultaneously: the packet is sent through the appropriate port and $\bot$ is sent through ACancelOut.

$$\lambda(s) = \begin{cases} \{E'\} \cup \{\bot\} & \text{if } s = (\texttt{true}, \texttt{false}, E') \\ \{E\} \cup \{\bot\} & \text{if } s = (\texttt{false}, \texttt{true}, E) \end{cases}.$$

The internal transition function resets the state to s_0.

It is important to note that the semantics of the Selector is well defined, even in a parallel setup. If a packet is received from each of ASuccessIn and AFailIn at the same time,[11] the output function is undefined. Hence no output is generated. However, if the Selector is a sub-component of an enclosing CRule, the `select` function prevents this situation from occurring. Consider, for example, the pattern in Fig. 10 and assume that the packet received through the CRule is matched by the two ARules Up and Down. The `select` function will choose one of them, say Up, to execute and output the resulting packet (step 2). The Selector will then receive the new packet and is thus ready to output. The imminent set consists subsequently of the Selector and Down. By step 1 of the `select` function, the Selector outputs the new packet as well as the cancel event $\bot$ which passivates Down. A more detailed trace of execution involving this pattern is provided in appendix B.

[11] Remember that the transformations are in-place and hence there is a single graph (often referred to as repository). Nevertheless, in a parallel setup, there may be multiple references to the graph in the flow at the same time.

A.4 The Synchronizer

The Synchronizer is yet another atomic DEVS, parametrized by the number of threads θ to synchronize:

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

The state S of the Synchronizer is composed of two integers t and f, counting the number of events received from ASuccessIn and AFailIn, respectively. π can hold an event; thus a state $s \in S$ can hold at most one event. The trivial case where $\theta = 1$ is legitimate, but at run-time, the Synchronizer becomes an overhead.

$$S = \left\{ (t, f, \theta, \pi) \,|\, t, f, \theta \in \mathbb{N}, \theta \geq 1, \pi \in \texttt{INSTANCEOF}\,(\text{Event}) \cup \{\emptyset\} \right\}.$$

S is initially set to $s_0 = (0, 0, \theta, \emptyset)$.

The ports are similar to those of the Selector, but with no port for cancelling:

$$\begin{aligned}
X &= X_{\texttt{ASuccessIn}} \times X_{\texttt{AFailIn}},\\
Y &= Y_{\texttt{ASuccessOut}} \times Y_{\texttt{AFailOut}}.
\end{aligned}$$

The time-advance returns 0 only when the number of events received reaches the threshold θ. In practice, this means that all threads have reached the Synchronizer.

$$\tau(s) = \begin{cases} 0 & \text{if } t + f = \theta \\ +\infty & \text{otherwise} \end{cases}, \forall s \in S.$$

After the application of δ_{ext}, π holds the latest event received from ASuccessIn. If no such event has been received yet, then it holds the first event received from AFailIn. The reception of simultaneous inputs is resolved similarly to the case of the Selector.

$$\delta_{\text{ext}}((s, e), x) = \begin{cases} (t+1, f, \theta, E') & \text{if } x = E' \\ (t, f+1, \theta, E') & \text{if } x = E \wedge \pi = E' \\ (t, f+1, \theta, E) & \text{if } x = E \wedge (\pi = E \vee \pi = \emptyset) \end{cases}.$$

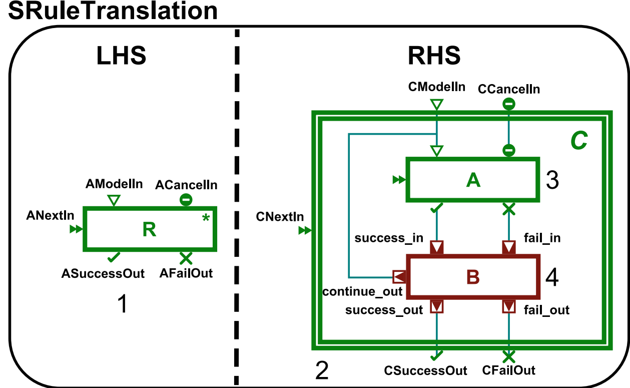


Fig. 24 An SRule is transformed to a CRule enclosing an ARule and an accumulator, which defines its semantics. The names *R*, *A*, and *B* are not part of the rule

The output function is only defined if at least one event has been received.

$$\lambda(s) = \begin{cases} \pi & \text{if } t \geq 1 \vee f \geq 1 \\ \phi & \text{otherwise} \end{cases}.$$

The internal transition function resets the state to s_0. The behaviour of the Synchronizer is very similar to the Selector's. The difference is that the latter outputs the event as soon as it has received it, whereas the former waits until a fixed number of events is received first.

A.5 The SRule

An SRule R is defined by the transformation rule of Fig. 24. MoTif internally translates R into a CRule C composed of an ARule A and an *accumulator* B. Therefore, to complete the formal definition of R, we define C and B. C is a CRule where $M = \{A, B\}$, $N = \{1, 2\}$ labels the A and B, respectively, and $I = \{I_1 = \{c, 2\}, I_2 = \{1\}, I_c = \{2\}\}$.

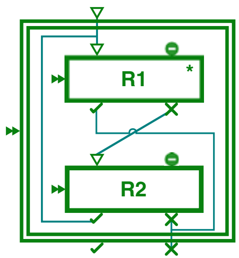
On the other hand, B has a lot of similarities with the Synchronizer. The state S is modified because we do not need to count successes and failures; thus t and f are booleans and θ is discarded. Another boolean α is used to indicate whether B is active (similar to the ARule).

$$S = \Big\{ (t, f, \alpha, \pi) \,|\, t, f, \alpha \in \{\texttt{true}, \texttt{false}\}, \\ \pi \in \texttt{INSTANCEOF}(\text{Event}) \cup \{\emptyset\} \Big\}.$$

It is initially set to $s_0 = (\texttt{false}, \texttt{false}, \texttt{false}, \emptyset)$.

The input set is the same as the Synchronizer. For the output set, we denote the event E'' sent from success_out, although it is the same as E, the one received from fail_in and thus $E'' = E$. The event received from success_in is sent via continue_out.

Fig. 25 Priorities encoded with SRules



$$X = X_{\texttt{success_in}} \times X_{\texttt{fail_in}},$$
$$X_{\texttt{success_in}} = \{E'\} \cup \{\phi\},$$
$$X_{\texttt{fail_in}} = \{E\} \cup \{\phi\},$$
$$Y = Y_{\texttt{success_out}} \times Y_{\texttt{fail_out}} \times Y_{\texttt{continue_out}},$$
$$Y_{\texttt{success_out}} = \{E''\} \cup \{\phi\},$$
$$Y_{\texttt{fail_out}} = \{E\} \cup \{\phi\},$$
$$Y_{\texttt{continue_out}} = \{E'\} \cup \{\phi\}.$$

The time-advance returns 0 only when an event is received.

$$\tau(s) = \begin{cases} 0 & \text{if } \alpha = \texttt{true} \\ +\infty & \text{otherwise} \end{cases}, \forall s \in S.$$

The reception of an event activates B.

$$\delta_{\text{ext}}((s, e), x) = \begin{cases} (\texttt{true}, f, \texttt{true}, x) & \text{if } x = E' \\ (t, \texttt{true}, \texttt{true}, x) & \text{if } x = E \end{cases}.$$

The output function outputs the event B received when it is active. If it was received through success_in, it is output via continue_out. If it was received through fail_in, it is output via success_out if B had previously received an event through success_in; otherwise, it is output via fail_out.

$$\lambda(s) = \begin{cases} E' & \text{if } \alpha = \texttt{true} \wedge f = \texttt{false} \\ E'' & \text{if } \alpha = \texttt{true} \wedge t = \texttt{true} \wedge f = \texttt{true} \\ E & \text{if } \alpha = \texttt{true} \wedge t = \texttt{false} \wedge f = \texttt{true} \\ \phi & \text{otherwise} \end{cases}.$$

The internal transition function passivates B:

$$\delta_{\text{int}}(t, f, \alpha, \pi) \\ = \begin{cases} (t, f, \texttt{false}, \pi) & \text{if } f = \texttt{false} \\ (\texttt{false}, \texttt{false}, \texttt{false}, \emptyset) & \text{otherwise} \end{cases}.$$

The SRule allows to easily encode priorities in a transformation as depicted in Fig. 25. Suppose that two rules $\mathbf{r}_1$ and $\mathbf{r}_2$ are such that $\mathbf{r}_1$ has higher priority. Then, R_1 would be an SRule encoding $\mathbf{r}_1$ and R_2 would be an ARule encoding $\mathbf{r}_2$. The failure outport of R_1 would be connected to the graph inport of R_2 and the success outport of R_2 to the one of R_1. Both the success outport of R_1 and the fail outport of R_2 will be connected to subsequent rule blocks.

Fig. 26 An FRule is transformed to a CRule enclosing an ARule and an accumulator, which defines its semantics. The names *R*, *A*, and *B* are not part of the rule

A.6 The FRule

An $\mathsf{FRule}_{r,\Delta,\sigma_1,\sigma_2}$ has the same structure as an ARule. The difference lies in the operational semantics: when the rule r is in $\Sigma^\top(\mathtt{r})$ with n possible matches, it consumes all of these matches by applying the transformation on each. Thus, the output γ' of $\lambda(s)$ results in a graph where r has been applied n times. The rule matching and application is atomic: therefore no ordering of the sequence of application may be assumed.

Alternatively, an FRule can be defined similarly to an SRule as in Fig. 26. The difference is that the continue_out port of B is connected to the ANextIn port of A. An FRule R is thus equivalent to a CRule C enclosing the corresponding ARule A and an accumulator B. B has two inports success_in and fail_in, respectively, connected from the ASuccessOut and AFailOut ports of A. Three outports are also needed for B: success_out, fail_out, and continue_out, respectively, connected to CSuccessOut of C, CFailOut of C, and ANextIn of A. If A outputs through ASuccessOut, B keeps track of a success and sends back the packet via continue_out. If A outputs through AFailOut and B is in success, then B outputs the packet via *success_out*. If A outputs through AFailOut and B is not in success, then B outputs the packet via fail_out.

Assuming multiple Pacman instances are allowed in the game, both Kill and Eat blocks could be turned into an FRule. This would encode that all Ghosts that can kill a PacMan do so atomically (in one step). Similarly, all PacMen that can eat a Pellet do so atomically.

A.7 The XRule

An $\mathsf{XRule}_{r,\Delta,\sigma_1,\sigma_2}$ is introduced to support backtracking. It has the same signature as an ARule, but the state S is extended with a stack Π of packets such as

$$S=\left\{(\gamma,\rho,\alpha,\sigma_1,\sigma_2,\Sigma(r),\Pi)\,|\,\gamma\in G^*,\rho=m(\gamma),\alpha,\sigma_1,\sigma_2\in\{\mathtt{true},\ \mathtt{false}\}\right\}.$$

The initial state of S and internal transition function are modified accordingly:

$$s_0=\left(\mathtt{nil},\mathtt{nil},\mathtt{false},\sigma_1,\sigma_2,\Sigma^i(r),\emptyset\right),$$
$$\delta_{\text{int}}(\gamma,\rho,\epsilon,\sigma_1,\sigma_2,\Sigma(r),\Pi)=(\gamma,\rho,\mathtt{false},\sigma_1,\sigma_2,\Sigma(r),\Pi).$$

The external transition function operates on Π with the usual stack predicates. $\Pi.\mathtt{PUSH}(x)$ returns Π with object x added on top of the stack. $\Pi.\mathtt{POP}()$ returns Π without the object at the top of the stack. $\Pi.\mathtt{PEEK}()$ returns the object currently at the top of the stack. If $\Pi=\emptyset$, the latter two operations return $\emptyset$.

$$\delta_{\text{ext}}((s,e),x)=\begin{cases}(\gamma,\rho,\mathtt{false},\sigma_1,\sigma_2,\Sigma(r),\Pi) & \\ \text{if } x=\bot\ \left(\gamma,\rho,\mathtt{true},\sigma_1,\sigma_2,\Sigma^M(r),\Pi.\mathtt{PUSH}\left(\overline{\langle\gamma,\rho\rangle}\right)\right) & \\ \quad \text{if } x=\langle\gamma,\rho\rangle\wedge\Sigma(r).\mathtt{INSTATE}\left(\Sigma^\top(r)\right) & \\ \left(\gamma,\rho,\mathtt{true},\sigma_1,\sigma_2,\Sigma^M(r),\Pi\right) & \\ \quad \text{if } x=\langle\gamma,\rho\rangle\wedge\Sigma(r).\mathtt{INSTATE}\left(\Sigma^\bot(r)\right) & \\ \left(\overline{\gamma},\overline{\rho},\mathtt{true},\sigma_1,\sigma_2,\Sigma^M(r),\Pi\right) & \\ \quad \text{if } x=E\wedge\Pi.\mathtt{PEEK}()=\langle\gamma,\rho\rangle\wedge\Sigma(r).\mathtt{INSTATE}\left(\Sigma^\top(r)\right) & \\ \left(\gamma,\rho,\mathtt{true},\sigma_1,\sigma_2,\Sigma^M(r),\Pi.\mathtt{POP}()\right) & \\ \quad \text{if } x=E\wedge\Pi.\mathtt{PEEK}()=\langle\gamma,\rho\rangle\wedge\Sigma(r).\mathtt{INSTATE}\left(\Sigma^\bot(r)\right) & \end{cases}.$$

The cancel case is identical to the one of ARule. When a packet is received from the AModelIn port, the rule goes into the matching mode Σ^M as previously. There are two cases. If a match is found, (i.e., r remains in $\Sigma^\top$), a copy[12] $\overline{\langle\gamma,\rho\rangle}$ of the packet is pushed on the stack Π. The state of the XRule, however, holds the original graph γ and pivot ρ. On the other hand, if the state of r is in $\Sigma^\bot$, the stack remains unchanged. Finally, we consider the cases where an event is received from the ANextIn port. If there is still at least one unprocessed match ($\Sigma(r).\mathtt{INSTATE}\left(\Sigma^\top(r)\right)$ is true), then the XRule enters the *roll-back* step. The state S then holds a copy of the packet on the top of the stack Π. That packet is also fed into rule r to process that match on the correct graph. The roll-back step is needed as the transformation is in-place and the graph may have been modified between the matching phase and when the next match is to be processed. The last case occurs if an event is received from the ANextIn port but there are no further matches. The packet on the top of the stack is placed in S to

[12] $\overline{x}$ denotes a "copy" of x.

ensure proper delivery of the original packet when the output function is called. The stack is then popped. Note, however, that if the stack was empty, we have $\delta_{\text{ext}}((s, e), x) = \left(\gamma, \rho, \texttt{true}, \sigma_1, \sigma_2, \Sigma^M(r), \emptyset\right)$, where γ and ρ remain unchanged from the previous state s. A concrete example illustrating the behaviour of an XRule is provided in Appendix B.

Appendix B: Execution trace

Tables 2, 3, 4, 5, and 6 present an execution trace of the transformation model in Fig. 9 and its sub-model in Fig. 10. This transformation models the behaviour of Pacman searching for Pellets to eat. The transformation encodes a pathfinding mechanism through back-tracking and recursion.

Table 2 The execution trace of the a host graph fed into SmartMove

Step	1	2	3
TryMove.Left	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	–
TryMove.Right	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\top, \langle\gamma_0, \emptyset\rangle)$	–
TryMove.Up	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	–
TryMove.Down	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\top, \langle\gamma_0, \emptyset\rangle)$	$(\gamma_1, \texttt{nil}, \texttt{false}, \Sigma^\perp, \langle\gamma_0, \emptyset\rangle)$
TryMove.Selector	$(\texttt{false}, \texttt{false}, \emptyset)$	–	$(\texttt{true}, \texttt{false}, \langle\gamma_1, \emptyset\rangle)$
Eat	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i)$	–	–
MakeMove.Left	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i, \emptyset)$	–	–
MakeMove.Right	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i, \emptyset)$	–	–
MakeMove.Up	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i, \emptyset)$	–	–
MakeMove.Down	$(\texttt{nil}, \texttt{nil}, \texttt{false}, \Sigma^i, \emptyset)$	–	–
MakeMove.Selector	$(\texttt{false}, \texttt{false}, \emptyset)$	–	–
t	0	–	1
γ		–	

Table 3 The execution trace of the a host graph fed into SmartMove

Step	4	5	6
TryMove.Left	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	–
TryMove.Right	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\top, \langle\gamma_0, \emptyset\rangle)$	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\top, \langle\gamma_0, \emptyset\rangle)$	$(\gamma_3, \texttt{nil}, \texttt{false}, \Sigma^\perp, \langle\gamma_0, \emptyset\rangle)$
TryMove.Up	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	–
TryMove.Down	–	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	–
TryMove.Selector	$(\texttt{false}, \texttt{false}, \emptyset)$	–	$(\texttt{true}, \texttt{false}, \langle\gamma_3, \emptyset\rangle)$
Eat	$(\gamma_1, \texttt{nil}, \texttt{true}, \Sigma^\perp)$	$(\gamma_1, \texttt{nil}, \texttt{false}, \Sigma^\perp)$	–
MakeMove.Left	–	–	–
MakeMove.Right	–	–	–
MakeMove.Up	–	–	–
MakeMove.Down	–	–	–
MakeMove.Selector	–	–	–
t	–	2	3
γ	–	(roll-back)	

Table 4 The execution trace of the a host graph fed into SmartMove

Step	7	8	9
TryMove.Left	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$
TryMove.Right	–	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$
TryMove.Up	–	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$
TryMove.Down	–	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$
TryMove.Selector	$(\texttt{false}, \texttt{false}, \emptyset)$	–	$(\texttt{false}, \texttt{true}, \langle\gamma_0, \emptyset\rangle)$
Eat	$(\gamma_3, \texttt{nil}, \texttt{true}, \Sigma^\perp)$	$(\gamma_3, \texttt{nil}, \texttt{false}, \Sigma^\perp)$	–
MakeMove.Left	–	–	–
MakeMove.Right	–	–	–
MakeMove.Up	–	–	–
MakeMove.Down	–	–	–
MakeMove.Selector	–	–	–
t	–	4	5
γ	–	(roll-back)	–

Table 5 The execution trace of the a host graph fed into SmartMove

Step	10	11	12
TryMove.Left	–	–	$(\gamma_6, \texttt{nil}, \texttt{true}, \Sigma^\top, \langle\gamma_6, \emptyset\rangle)$
TryMove.Right	–	–	$(\gamma_6, \texttt{nil}, \texttt{true}, \Sigma^\top, \langle\gamma_6, \emptyset\rangle)$
TryMove.Up	–	–	$(\gamma_6, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$
TryMove.Down	–	–	$(\gamma_6, \texttt{nil}, \texttt{true}, \Sigma^\top, \langle\gamma_6, \emptyset\rangle)$
TryMove.Selector	$(\texttt{false}, \texttt{false}, \emptyset)$	–	–
Eat	–	–	–
MakeMove.Left	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	–	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$
MakeMove.Right	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\top, \langle\gamma_0, \emptyset\rangle)$	$(\gamma_6, \texttt{nil}, \texttt{false}, \Sigma^\perp, \langle\gamma_0, \emptyset\rangle)$	–
MakeMove.Up	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\perp, \emptyset)$	–	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\perp, \emptyset)$
MakeMove.Down	$(\gamma_0, \texttt{nil}, \texttt{true}, \Sigma^\top, \langle\gamma_0, \emptyset\rangle)$	–	$(\gamma_0, \texttt{nil}, \texttt{false}, \Sigma^\top, \langle\gamma_0, \emptyset\rangle)$
MakeMove.Selector	–	$(\texttt{true}, \texttt{false}, \langle\gamma_6, \emptyset\rangle)$	–
t	–	6	–
γ	–		–

The host model is represented in Table 2 at step 1. In the state of the ARule and XRules, σ_1, and σ_2 are set to `false` and are omitted for conciseness. We also assume that for every ARule and XRule, $\Delta = 1$. The notation $\langle\gamma_i, \rho_i\rangle$ denotes the packet created at time $t = i$. The transition from step 2 to step 3 follows the internal transition of TryMove.Down, having been selected by the select function. The transition from step 3 to step 5 follows the internal transition of TryMove.Selector. The models shown in the last row of the tables indicate the new graph γ after a successful application of a transformation rule. The symbol "-" indicates that the content of the current cell is the same as the cell on the same row in the previous column.

Table 6 The execution trace of the a host graph fed into SmartMove

Step	13	14	15
TryMove.Left	–	$(\gamma_6, \texttt{nil}, \texttt{false}, \Sigma^{\top}, \langle\gamma_6, \emptyset\rangle)$	–
TryMove.Right	–	$(\gamma_6, \texttt{nil}, \texttt{false}, \Sigma^{\top}, \langle\gamma_6, \emptyset\rangle)$	–
TryMove.Up	–	$(\gamma_6, \texttt{nil}, \texttt{false}, \Sigma^{\perp}, \langle\gamma_6, \emptyset\rangle)$	–
TryMove.Down	$(\gamma_7, \texttt{nil}, \texttt{false}, \Sigma^{\perp}, \langle\gamma_6, \emptyset\rangle)$	–	–
TryMove.Selector	$(\texttt{true}, \texttt{false}, \langle\gamma_7, \emptyset\rangle)$	$(\texttt{false}, \texttt{false}, \emptyset)$	–
Eat	–	$(\gamma_7, \texttt{nil}, \texttt{true}, \Sigma^{\top})$	$(\gamma_8, \texttt{nil}, \texttt{false}, \Sigma^{\perp})$
MakeMove.Left	–	–	–
MakeMove.Right	–	–	–
MakeMove.Up	–	–	–
MakeMove.Down	–	–	–
MakeMove.Selector	–	–	–
t	7	–	8
γ	Score: 0	–	Score: 1

The termination of MoTif transformations is ensured by the *termination condition* of the DEVS simulator which is

- All atomic models are passivated, (i.e., $\tau(s) = +\infty$); or
- Specified explicitly, (e.g., time or state condition).

References

1. Blostein, D., Fahmy, H., Grbavec, A.: Issues in the practical use of graph rewriting. In: Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G. (eds.) Selected Papers from the 5th International Workshop on Graph Grammars and Their Application to Computer Science. LNCS, vol. 1073, pp. 38–55. Springer, Williamsburg, November 1996
2. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: ICSE'00, pp. 742–745. ACM Press, Limerick, June 2000
3. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Control flow support in metamodel-based model transformation frameworks. In: EUROCON'05, pp. 595–598. IEEE, Belgrade, November 2005
4. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. SoSym **5**(3), 261–288 (2006)
5. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model transformation with a visual control flow language. IJCS **1**(1), 45–53 (2006)
6. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with PROGRES. In: Schäfer, W., Botella, P. (eds.) 5th European Software Engineering Conference. LNCS, vol. 989, pp. 219–234. Springer, Sitges, September 1995
7. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: a standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture—Foundations and Applications: Second European Conference. LNCS, vol. 4066, pp. 361–375. Springer, Berlin (2006)
8. 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. ENTCS, vol. 148, no. 1, pp. 187–198. Elsevier (2006)
9. de Lara, J., Vangheluwe, H.: AToM^3: a tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE'02. LNCS, vol. 2306, pp. 174–188. Springer, Grenoble, April 2002
10. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G.: Handbook of graph grammars and computing by graph transformation. In: Rozenberg, G. (ed.) Foundations, vol. 1. World Scientific Publishing Co. (1997)
11. Zeigler, B.P.: Multifacetted Modelling and Discrete Event Simulation. Academic Press, New York (1984)
12. Xie, H., Boukerche, A., Zhang, M., Zeigler, B.P.: Design of a QoS-aware service composition and management system in peer-to-peer network aided by DEVS. In: DS-RT, pp. 285–291 (2008)
13. Lee, J.-K., Lim, Y.-H., Chi, S.-D.: Hierarchical modeling and simulation environment for intelligent transportation systems. Simulation **80**(2), 61–76 (2004)
14. Filippi, J.-B., Bisgambiglia, P.: JDEVS: an implementation of a DEVS based formal framework for environmental modelling. Environ. Model. Softw. **19**(3), 261–274 (2004)
15. Bolduc J.-S., Vangheluwe, H.: The modelling and simulation package pythonDEVS for classical hierarchical DEVS. McGill University. MSDL Technical Report MSDL-TR-2001-01, June 2001
16. Mens, T., Van Gorp, P.: A taxonomy of model transformation. In: GraMoT'05. ENTCS, vol. 152, pp. 125–142, Tallinn (Estonia), March 2006
17. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with DEVS. In: Nagl, M., Schürr, A. (eds.) AGTIVE'07. LNCS, vol. 5088, pp. 136–152. Springer, Kassel (2007)
18. Guerra, E., de Lara, J.: Event-driven grammars: relating abstract and concrete levels of visual languages. SoSym **6**(6), 317–347 (2007)
19. Guerra, E., de Lara, J.: Event-driven grammars: towards the integration of meta-modelling and graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT'04. LNCS, vol. 3256, pp. 54–69. Springer, New York (2004)

20. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(3), 100–107 (1968)
21. Zaitsev, A.V., Skorik, Y.A.: Mathematical description of sensorimotor reaction time distribution. Human Physiol. **28**(4), 494–497 (2002)
22. Devroye, L.: Non-Uniform Random Variate Generation. Springer, New York (1986)
23. Gyapay, S., Heckel, R., Varró, D.: Graph transformation with time: causality and logical clocks. In: ICGT'02. LNCS, vol. 2505, pp. 120–134. Springer, Barcelona, October 2002
24. Heckel, R., Lajios, G., Menge, S.: Stochastic graph transformation systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT'04. LNCS, vol. 3256, pp. 243–246. Springer, New York (2004)
25. Agrawal, A.: Metamodel based model transformation language. In: OOPSLA'03, pp. 386–387. ACM Press, Anaheim (2003)
26. Vizhanyo, A., Agrawal, A., Shi, F.: Towards generation of efficient transformations. In: Karsai, G., Visser, E. (eds.) GPCE'04. LNCS, vol. 3286, pp. 298–316. Springer, New York (2004)
27. http://www.isis.vanderbilt.edu/projects/gme/. 6 Dec 2008
28. Blostein, D., Schürr, A.: Computing with graphs and graph rewriting. SPE **9**(3), 1–21 (1999)
29. Zündorf, A.: Graph pattern matching in PROGRES. In: Ehrig, H., Engels, G., Rozenberg, G. (eds.) Graph Grammars and Their Application to Computer Science. LNCS, vol. 1073, pp. 454–468. Springer, Williamsburg, November 1994
30. Zündorf, A.: Implementation of the imperative/rule based language PROGRES. Department of Computer Science III, Aachen University of Technology, Germany, Aachener Informatik-Berichte 92-38 (1992)
31. Fischer, T., Niere, J., Turunski, L., Zündorf, A.: Story diagrams: a new graph rewrite language based on the Unified Modelling Language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Theory and Application of Graph Transformations. LNCS, vol. 1764, pp. 296–309. Springer, Paderborn, November 2000
32. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In: Proceedings of the 27th International Conference on Software Engineering ICSE '05, pp. 670–671. ACM, New York (2005)
33. Henkler, S., Greenyer, J., Hirsch, M., Schäfer, W., Alhawash, K., Eckardt, T., Heinzemann, C., Loffler, R., Seibel, A., Giese, H.: Synthesis of timed behavior from scenarios in the Fujaba Real-Time Tool Suite. In: ICSE '09, pp. 615–618. IEEE Computer Society (2009)
34. Schürr, A.: Specification of graph translators with triple graph grammars. In: Tinhofer, G. (ed.) Graph-Theoretic Concepts in Computer Science. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg, June 1994
35. 4th International Workshop on Graph-Based Tools: The Contest. September 2008/07/21. [Online]. http://www.fots.ua.ac.be/events/grabats2008/
36. Taentzer, G.: AGG: a graph transformation environment for modeling and validation of software. In: AGTIVE'03. LNCS, vol. 3062, pp. 446–453. Springer, New York (2004)
37. Muliawan, O., Schippers, H., Van Gorp, P.: Model driven, Template based, Model Transformer (MoTMoT). http://motmot.sourceforge.net (2005)
38. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3), 214–234 (2007)
39. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with time for simulation-based design. In: Pierantonio, A., Vallecillo, A., Bézivin, J., Gray, J. (eds.) ICMT'08. LNCS, vol. 5063, pp. 91–106. Springer, Zürich, July 2008
40. Syriani, E.: A multi-paradigm foundation for model transformation language engineering. Ph.D. Thesis, McGill University, February 2011
41. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT'08. LNCS, vol. 5063, pp. 107–121. Springer, New York (2008)
42. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: MoDELS'06. LNCS, pp. 321–335 (2006)
43. Feng, T.H.: Model transformation with hierarchical discrete-event control. Ph.D. Thesis, EECS Department, University of California, Berkeley, USA, May 2009
44. Simulink User's Guide. MathWorks, Natick, USA. March 2010
45. Dormoy, F.X.: SCADE 6: A Model Based Solution for Safety Critical Software Development. Esterel Technologies, Toulouse (2007)
46. Chow, A.C.-H., Zeigler, B.P.: Parallel DEVS: a parallel, hierarchical, modular modeling formalism. TSCS **13**, 55–67 (1996)
47. Hunicke, R.: The case for dynamic difficulty adjustment in games. In: Proccedings of the ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, pp. 429–433. ACM, Valencia (2005)

Author Biographies

Eugene Syriani is currently a researcher in the School of Computer Science, McGill University. He received his B.Sc. in Mathematics and Computer Science in 2006 and his Ph.D. in Computer Science in 2011, both obtained at McGill University. He is a member of the Modeling, Simulation, and Design Lab supervised by Prof. Hans Vangheluwe. His academic work is sponsored by the Natural Sciences and Engineering Research Council of Canada. His current research interests are Model Transformation, Model-driven Engineering, and Simulation-based Design. He is particularly interested in the engineering of model transformation languages. His contribution in the field resides in the engineering of model transformation languages, following multi-paradigm modelling principles. He has developed a framework for producing transformation languages tailored for the specific needs, based on T-Core. He also has over five years of industry experience in different service-oriented software companies in Montreal, Canada.

Hans Vangheluwe is a Professor in the Department of Mathematics and Computer Science, Antwerp University, Belgium, an Associate Professor in the School of Computer Science, McGill University, Montreal, Canada, and an Adjunct Professor at the National University of Defense Technology (NUDT), Changsha, China. He holds a D.Sc. degree, as well as M.Sc. degrees in Computer Science and in Theoretical Physics, all from Ghent University, Belgium. He heads the Modelling, Simulation and Design (MSDL) research lab, geographically distributed over McGill and Antwerp University. He has been the Principal Investigator of a number of research projects focussed on the development of a multi-formalism theory and enabling technology for Modelling and Simulation. Some of this work has led to the WEST++ tool, which was commercialised for use in the design and optimization of bioactivated sludge Waste Water Treatment Plants. He was the co-founder and coordinator of the European Union's ESPRIT Basic Research Working Group 8467 "Simulation in Europe", a founding member of the Modelica (http://www.modelica.org) Design Team, and an advisor to national and international granting agencies in Europe and North America. In a variety of projects, often with industrial partners, he applies the model-based theory and techniques of Computer Automated Multi-Paradigm Modelling (CAMPaM) in a variety of application domains. He has published over 100 peer-reviewed papers. He is an Associate Editor of the International Journal of Critical Computer-Based Systems, of Simulation: Transactions of the Society for Computer Simulation, and of the International Journal of Adaptive, Resilient and Autonomic Systems. His current interests are in domain-specific modelling and simulation, including the development of graphical user interfaces for multiple platforms. The MSDL's tool AToM3 (A Tool for Multi-formalism and Meta-Modelling), developed in collaboration with Prof. Juan de Lara uses meta-modelling and graph transformation to specify and generate domain-specific environments. Recently, he has become active in the design of Automotive applications.