Programmed Graph Rewriting with Time for Simulation-Based Design

Eugene Syriani and Hans Vangheluwe

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

Abstract. The Discrete EVent system Specification (DEVS) formalism allows for highly modular, hierarchical modelling of timed, reactive systems. DEVS can be used to describe complex control structures for programmed graph transformation. A side-effect of this approach is the introduction of an explicit notion of time. In this paper we 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 with programmed graph transformation based on DEVS. 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 vary 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.

1 Introduction

Programmed (or structured) graph transformation is one of the keys to making graph transformation scalable (and hence industrially applicable). Tools such as FUJABA [1], GReAT [2], VMTS [3], PROGReS [4], and MOFLON [5] support programmed graph transformation. These tools mostly introduce their own control flow language. In [6] we have shown the advantages of re-using a discrete-event modelling/simulation formalism to describe transformation control. In this paper, we will focus on the time aspect of modelling complex transformations, a side-effect of using a discrete-event modelling formalism. This is done by means of the well-known PacMan example, presented in Section 2. Section 3 introduces the DEVS formalism and how it is used for structured graph transformation. Section 4 describes how not only the PacMan game, but also the player can be explicitly modelled. Section 5 describes game simulation experiments in detail. Finally, Section 6 summarizes, concludes and proposes future work.

A. Vallecillo, J. Gray, A. Pierantonio (Eds.): ICMT 2008, LNCS 5063, pp. 91–106, 2008.

2 Case Study: The PacMan Game

To demonstrate the power of timed, programmed graph transformation, in particular in the context of simulation-based design, we use a simplified version

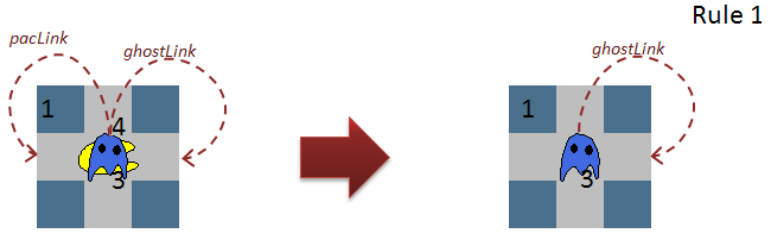

Fig. 1. PacMan Semantics: Ghost kills PacMan rule

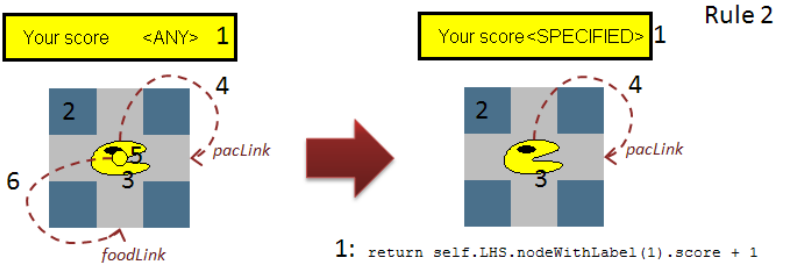

Fig. 2. PacMan Semantics: PacMan eats Food rule

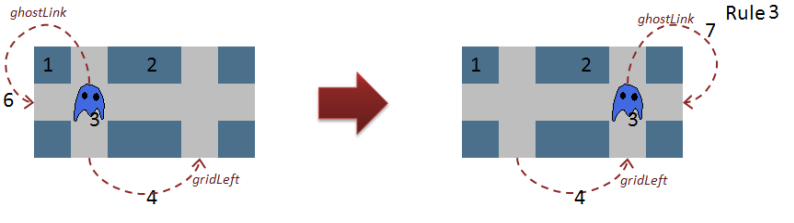

Fig. 3. PacMan Semantics: Ghost moves right rule

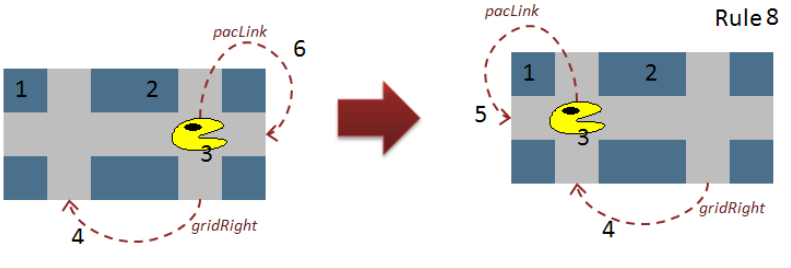

Fig. 4. PacMan Semantics: PacMan moves left rule

of the PacMan video game inspired by Heckel's tutorial introduction of graph transformation [7].

2.1 The PacMan Language (Abstract and Concrete Syntax)

The PacMan language has five distinct elements: PacMan, Ghost, Food (Pellets), GridNode and ScoreBoard. PacMan, Ghost and Food objects can be linked to GridNode objects. This means that these objects can be "on" a GridNode. GridNode objects are geometrically organized in a grid, similar to the PacMan video game. Adjacency implies that PacMan and Ghost "may move" to a connected GridNode. A ScoreBoard object holds an integer valued attribute score. Our tool AToM^3 [8] allows modelling of both abstract and visual concrete syntax (including geometric/topological constraint relations such as a PacMan being centered over a GridNode). From these models, an interactive, visual PacMan modelling environment is synthesized.

2.2 The PacMan Semantics (Graph Transformation)

The operational semantics of the PacMan formalism is defined in a Graph Transformation model which consists of a number of rules. In the rules in the following figures, concrete syntax is used. This is a useful feature for domain-specific modelling specific to AToM^3. Dashed lines were added to explicitly show the "on" links. Rule 1 in Figure 1 shows killing: when a Ghost object is on a GridNode which has a PacMan object, the PacMan is removed. Rule 2 in Figure 2 shows eating: when a PacMan object is on a GridNode which has a Food object, Food is removed and the score gets updated (using an attribute update expression). Note how in the sequel, we will focus on game playability and will ignore the score. Rule 3 in Figure 3 expresses the movement of a Ghost object to the right and rule 8 in Figure 4 the movement of a PacMan object to the left. Similar rules to move Ghosts and PacMan objects up, down, left and right are not shown. Rules 1 and 2 have priorities 1 and 2 respectively. All remaining rules have the same priority 3.

3 DEVS for Programmed Graph Transformation

We previously [6] demonstrated how the *Discrete EVent system Specification* (DEVS) formalism can be used as a semantic domain for Programmed Graph Transformation. In this section, our approach is described, elaborating on the the implementation in AToM^3 of the ideas introduced in [6]). This description will form the basis for following sections which will focus on the use of time in our models.

3.1 The Discrete Event System Specification (DEVS)

The DEVS formalism was introduced in the late seventies by Bernard Zeigler as a rigorous basis for the compositional modelling and simulation of discrete event systems [9].

A DEVS model is either *atomic* or *coupled*. An atomic model describes the behaviour of a reactive system. A coupled model is the composition of several DEVS sub-models which can be either atomic or coupled. Submodels have *ports*, which are connected by channels. Ports are either *input* or *output*. Ports and channels allow a model to send and receive signals (events) between 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.

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 specified by the *time-advance* of that state or until input is received on some port. If no input is received, after the time-advance of the state expires, the model first (before changing state) sends output, specified by the *output function* and then instantaneously jumps to a new state specified by the *internal transition function*. If input is received before the time for the next internal transition however, then it is the *external transition function* which is applied. The external transition depends on the current state, the time elapsed since the last transition and the inputs from the input ports.

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

For this paper, we use our own DEVS simulator called `pythonDEVS` [10], grafted onto the object-oriented scripting language Python.

3.2 Controlled Graph Rewriting with DEVS

At the heart of our approach is the embedding of graphs in DEVS events and of individual transformation rules into *atomic*DEVS blocks. Figure 5 shows how our approach comprises a number of transformations. First, we model a collection of transformation rules in domain-specific 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 (`pacDie` is shown here on the top right). The core of the generated code is the method `execute` which takes a (host) graph as argument and encodes the transformation rule (matching and re-writing). Second, we build a hierarchical model of the Modelled and Modular Timed Graph Transformation language, in the MoTif (Modular Timed model transformation) visual modelling language (shown at the bottom left of the figure). All building

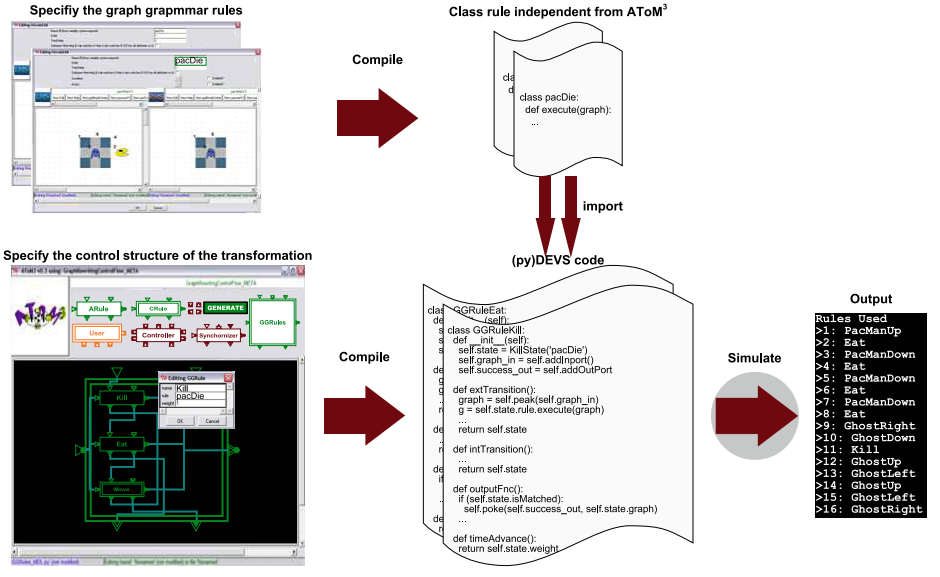

Fig. 5. DEVS-based Programmed Graph Rewriting Architecture

blocks have ports for incoming graphs (top left port), outgoing graphs in case of successful rule application (bottom left port), outgoing (unmodified) graphs in case of failed rule application (bottom right port), incoming transformation interrupt (top right port), incoming pivot (hint about where to start matching) information (left side port) and outgoing pivot information (right side port). These ports appear on both atomic (*ARule*: single rectangle frame) and coupled (*CRule*: double rectangle frame) transformation models which implies that they can be used interchangeably to build complex hierarchical transformation models. *ARules* contain a reference to the compiled rule class. Other special atomic models such as a **Synchronizer** block as well as default atomic and coupled models can be used to control the flow of the transformation. Third, the MoTif model gets compiled into a DEVS model. *CRules* get translated into *coupled*DEVS models. *ARules* models get translated into *atomic*DEVS models. In the latter, the `execute` method encoding the transformation is called in the `external transition` function of the *atomic*DEVS model. This transition function is triggered by the arrival of an external event (in which a to-be-transformed graph is embedded). Finally, all generated code is linked and presented to a DEVS simulator which performs the transformation and produces a trace.

3.3 The PacMan Case Study

The overall model of the PacMan game is shown in Figure 6.

The *coupled*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 the context of our previous work [6], the control information was in the form of key code presses to model the user interrupts of 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 **Autonomous Rules** *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 **User Controlled Rules** *CRule*). The **Autonomous Rules** *CRule* expects a graph to perform the rewriting, whereas the **User Controlled Rules** *ARule* waits for a control, too. The details are omitted here to focus on the overall structure.

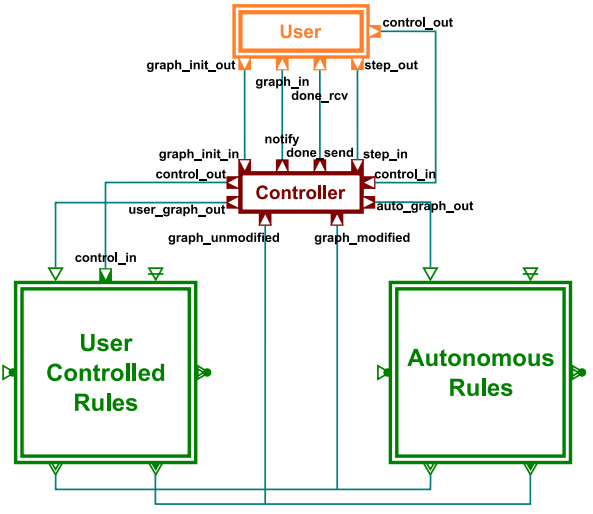

Fig. 6. Overall Transformation Model

The model described in [6] 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 the **Autonomous Rules** 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 how long a transformation takes. From the input model perspective, it gives a way of quantifying the complexity of a model. This does however not take into consideration any notion of game levels or any real-time behaviour which such a game 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 will present an extended model with focus on timing information. This will allow us, through simulation, to construct an optimally "playable" game.

4 Modelling Game *and* Player

The previous section showed how we can model both game syntax (using meta-modelling) and game dynamics (using programmed graph transformation) in an intuitive fashion suitable for non-software-experts. In our approach, the programmed graph transformation model gets compiled into a DEVS model which can subsequently be simulated.

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 between the display of subsequent steps is coded 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 Figure 6). Note that in this interaction model, time is explicitly present.

4.1 Modelling the Player

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 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 could mean different user reaction speed or different levels of decision analysis (such as pathfinding). 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 *performance metric*. Also, necessary data to calculate performance metrics needs to be automatically collected during experiments.

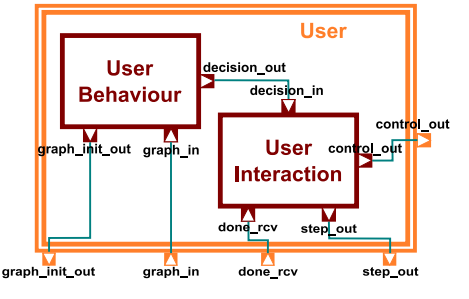

Fig. 7. Enhanced User Model

Explicitly modelling player behaviour can be done without modifying the overall model described in section 3.2 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 Figure 7.

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 **User Behaviour** *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 **User Behaviour** block models the actual behaviour of the player (often referred to as "AI" in the game community). It is this block which, after every transformation step, receives the new game state graph, analyzes it, and outputs a decision determining what the next game action (such as PacMan move up) will be. Also, since it is the **User Interaction** 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.

The notion of Event-driven Graph Rewriting [11] can be found in the literature. It was proposed in the context of a meta-modelling editor: a graph rewriting rule would be triggered in response to a user action. This concept is incorporated in the **User Controlled Rules** *coupled*DEVS block where a rule gets triggered depending on the user action. In our approach the user and user interaction itself has been modelled in the **User** *coupled*DEVS block.

Different players may use different *strategies*. Each strategy leads to a different model in the **User Behaviour** block. We have modelled three types of players for our experiments: Random, Dummy, 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 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, 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.

The Dummy 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 Dummy user. Whereas the Dummy user is restricted to making decisions based 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* [12]. Modelling A* with graph transformation rules requires backtracking and is outside the scope of this paper. Our prototype

implementation sidesteps the pathfinding problem by slightly modifying the meta-model of the PacMan formalism. Relative coordinates were added to the *gridNode* class with the condition that if a *gridNode* instance *g1* is associated with another instance *g2* via the *gridLeft* association, then $g1.x < g2.x$ and $g1.y = g2.y$. Similar conditions are defined for *gridRight*, *gridTop* and *gridBottom* associations. Therefore, the pathfinding only needs to compute the shortest Manhattan distance from PacMan to Food as well as a simple check for the grid node coordinates of the Ghost.

We compare the performance of different user behaviour types in Section 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 Dummy 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.

4.2 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 graph continues to loop between the **Controller** block and the **Autonomous Rules** block. If no instantaneous rule (**Kill** or **Eat**) matches, then it is the lower priority **Ghost Move** block that modifies the graph. In our earlier work [6], the graph received by this *CRule* was concurrently sent to the **Up**, **Down**, **Left** and **Right** *ARules* to make the ghost move. Non-deterministically, one of the matching rules got applied. This modelled a random movement of the Ghost. In order to generalize this behaviour to allow different strategies, a modification of the way the graph is sent to these *ARules* is necessary.

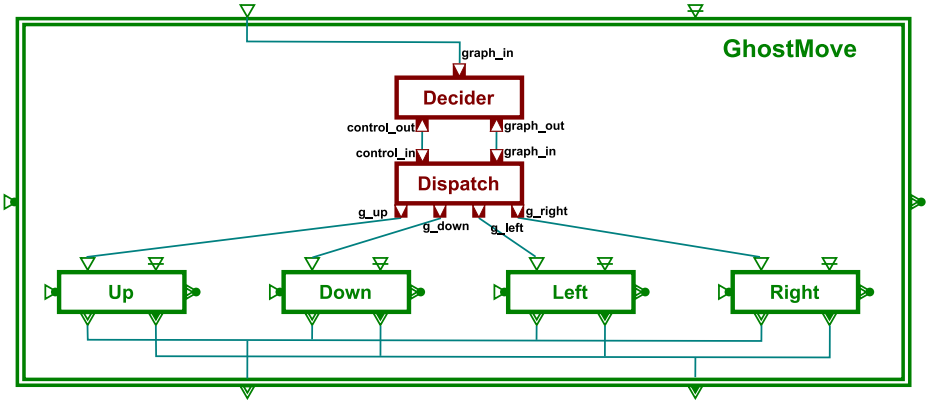

Fig. 8. Enhanced Ghost Behaviour Model

Figure 8 illustrates the modified topology of the Ghost movement model. The game state graph is received by a **Decider** *atomic*DEVS block. Similar to the **User Behaviour** block, it emits a direction that drives the movement of the Ghost. The Random, Dummy and Smart strategies are analogous to the player. The Random Ghost will randomly choose a direction, the Dummy 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. The same argument previously made about optimal pathfinding and backtracking applies. Then, 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 **User Controlled Rules** *CRule*.

4.3 Explicit Use of Time

We have now modelled both game and player, and the behaviour of both can use Random, Dummy, 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.

Timed Graph Transformation, as proposed by Gyapay, Heckel and Varró [13] 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 Timed Graph Transformation, it is the execution of a rule that can increase time and not the rule itself. Hence, the control flow (of the graph transformation) has full access to time. As pointed out in [13], time can be used as a metric to express how many time units are consumed to execute a rule. Having time at the level of the block containing a rule rather that in the rule itself does not lose this expressiveness.

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 of the PacMan grammar 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 **User Behaviour** 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 $5ms$). 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.

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 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 [14] give human reaction times (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 of frequencies 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 is; b characterizes the reaction speed of the subject. For simulation purposes, sampling from such a distribution is done by using the Inverse Cumulative Method.

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 [14].

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 Food is left on the board (victory). To appreciate these results, the score is also measured for each run. Simulations were run for a game configuration with 24 gridNodes, 22 Food Pellets, 1 Ghost and 1 PacMan. The game speed (ghost decision time) was varied from $100ms$ to $400ms$. 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 9 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 $310ms$ for the VeryFast user and $350ms$ for the Normal user), the curves tend to plateau. 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.

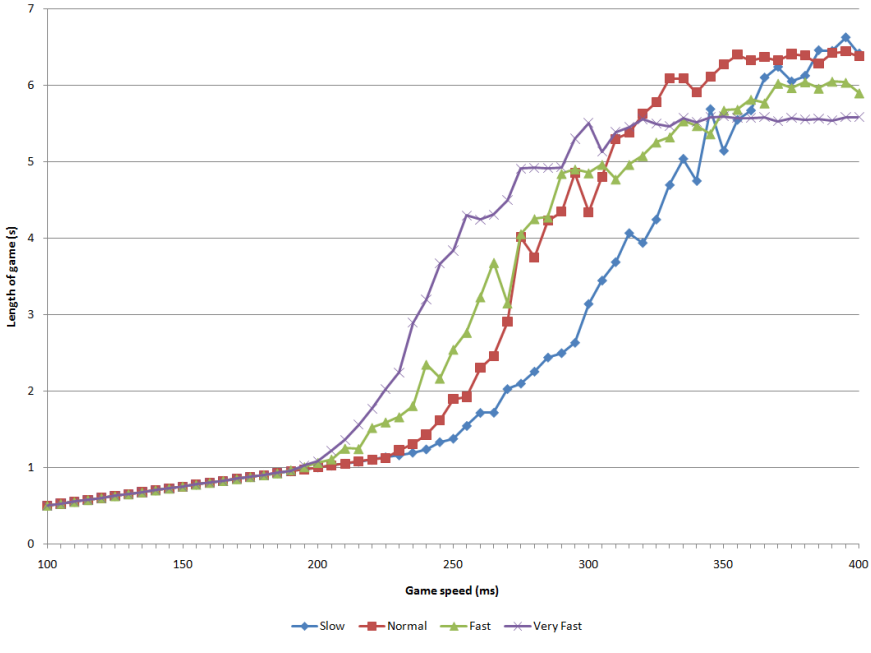

Fig. 9. Time till end

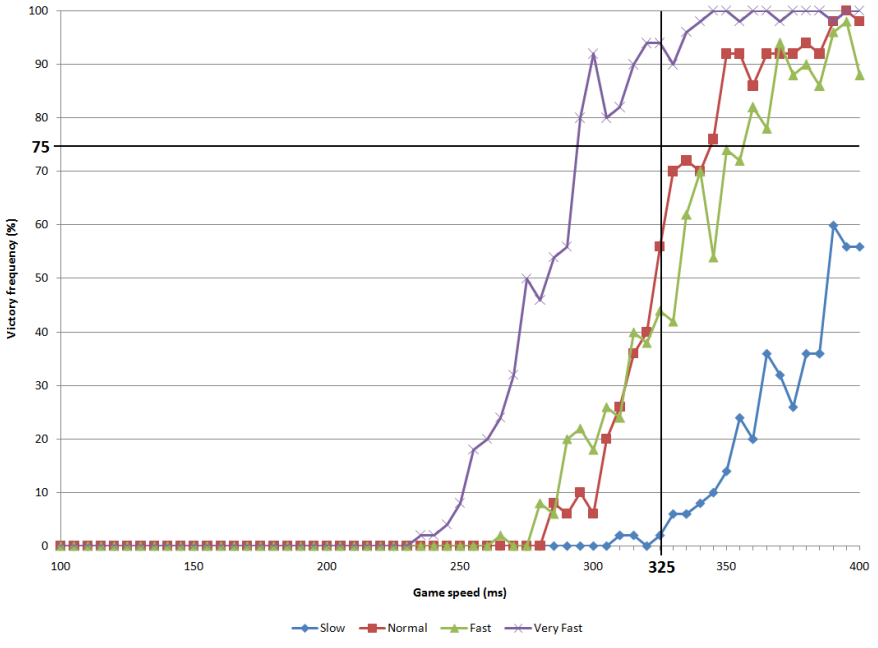

Fig. 10. Victory frequency

Figure 10 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 $325ms$.

To give further insight in the variability of the game experience, Figure 11 shows the *game length distribution* at the optimal *time advance* value. It is a unimodal distribution with a peak at $7.5s$. This average is quite low, but not surprising given the small game board. Experience with the finally 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 choosing the next movement of the ghost entity, we can now test the simulated game with real users, in real-time. We simply *discard the player model* and deploy the real-time game model (by executing the translated programmed graph rewriting system in a real-time DEVS simulator). In an attempt to generate the application completely from models, we (mostly) synthesized (yet another model transformation) an Ajax/SVG-based application from the PacMan meta-model built in AToM^3.

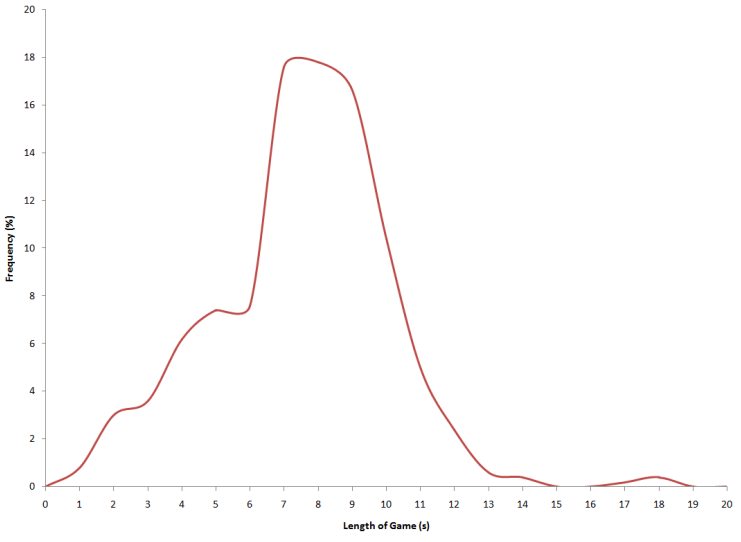

Fig. 11. Game length distribution; Normal user, game time advance $325ms$

6 Conclusions

In this article, we described the use of 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. 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.

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 decision about which next move the computer player (Ghost) should make was simplified by avoiding pathfinding concerns as mentioned in Section 4.1. We plan to investigate the specification of pathfinding strategies by means of graph transformation rules. This will require support for backtracking.

At the model structure level, it is noted how topologically similar the **User Controlled Rules** and **Ghost Move** *CRules* are. Re-use and parametrization of transformation models deserves further investigation.

Acknowledgments

The Canadian NSERC and MITACS agencies are gratefully acknowledged for partial support of this work.

References

1. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: ICSE 2000, pp. 742–745 (2000)
2. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. Software and Systems Modeling (SoSyM) 5, 261–288 (2005)
3. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model transformation with a visual control flow language. International Journal of Computer Science (IJCS) 1, 45–53 (2006)
4. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with PROGRES. In: Proceedings of the 5th European Software Engineering Conference, pp. 219–234. Springer, Heidelberg (1995)
5. Schürr, A., Rötschke, T., Amelunxen, C., Königs, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
6. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with DEVS. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE), pp. 134–149 (2007)
7. 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, pp. 187–198. Elsevier, Amsterdam (2006)
8. de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and metamodelling. In: Kutsche, R.-D., Weber, H. (eds.) ETAPS 2002 and FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
9. Zeigler, B.P.: Multifacetted Modelling and Discrete Event Simulation. Academic Press, London (1984)

10. Bolduc, J.S., Vangheluwe, H.: The modelling and simulation package PythonDEVS for classical hierarchical DEVS. MSDL technical report MSDL-TR-2001-01, McGill University (2001)
11. Guerra, E., de Lara, J.: Event-Driven Grammars: Towards the Integration of Metamodelling and Graph Transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 54–69. Springer, Heidelberg (2004)
12. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics 4, 100–107 (1968)
13. Gyapay, S., Heckel, R., Varró, D.: Graph transformation with time: Causality and logical clocks. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 120–134. Springer, Heidelberg (2002)
14. Zaitsev, A.V., Skorik, Y.A.: Mathematical description of sensorimotor reaction time distribution. Human Physiology 28(4), 494–497 (2002)