Animation of Tile-Based Games Automatically Derived from Simulation Specifications

Jan Wolter, Bastian Cramer, and Uwe Kastens

University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany
jwolter@mail.uni-paderborn.de,{bcramer,uwe}@uni-paderborn.de

Abstract. Visual Languages (VLs) are beneficial particularly for domain-specific applications, since they can support ease of understanding by visual metaphors. If such a language has an execution semantics, comprehension of program execution may be supported by direct visualization. This closes the gap between program depiction and execution. To rapidly develop a VL with execution semantics a generator framework is needed which incorporates the complex knowledge of simulating and animating a VL on a high specification level.
In this paper we show how a fully playable tile-based game is specified with our generator framework DEViL. We illustrate this on the famous Pac-man[1] game.
We claim that our simulation and animation approach is suitable for the rapid development process. We show that the simulation of a VL is easily reached even in complex scenarios and that the automatically generated animation is mostly adequate, even for other kinds of VLs like diagrammatic, iconic or graph based ones.

1 Introduction

A prominent representative of a visual language is the Unified Modeling Language (UML) [9] which is often used in software engineering process. Even smaller languages precoined for a specific domain are popular, because they can use visual metaphors of the target domain. In general an instance of such a visual language is used to produce source code of a different domain, e.g. Java Code from an UML class diagram.

To gain acceptance in rapid prototyping generator frameworks are used which can generate graphical structure editors for such visual languages from high-level specifications. These generators incorporate expert knowledge to produce a complete development environment for a VL with all features known from typical text editors like cut and paste, printing, drag and drop and so on. Unfortunately there is still a gap between program depiction and the generated code of that program. The programmer has to keep in mind what the program, he just created, does when it is executed. This gap is known as the gulf of execution [8]. Simulation and animation of the visual language instance can help to narrow

[1] Pac-man® is a registered trademark of Namco.

INForum 2010 - II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), 9-10 Setembro, 2010, pp. 113–124

this gap. The execution semantics of a visual language (if it has one) can be integrated into the visual language. Hence the VL instance is no longer static it can be simulated and smoothly animated. The user can "see" his language being executed before he generates code.

This helps to avoid mistakes at a very early stage and it supports program comprehension which is a challenging task especially in languages where many things happen in parallel.

The *Development Environment for Visual Languages*, DEViL, is a generator framework for visual languages which produces graphical editors from declarative high-level specifications. We extended it with simulation and animation support for VLs whereas a smooth and challenging animation can be derived automatically from a simple simulation specification. In this paper we want to show that our simulation specification language is powerful to simulate even complex behavior. We claim that the language helps in rapid prototyping, because simulation becomes an easy task due to powerful encapsulated concepts like event driven simulation and the extension of the simulation model. We will show that the automatically derived animation is suitable in most situations.

We will demonstrate this on the famous Pac-man game. It has a playful character, but it is also a challenging language for simulation, because of the complex navigation concepts of the "ghost" pawns in the game.

The paper is structured as follows: First we introduce the DEViL system and its underlying specification concepts with particular attention to simulation and animation. In Section 3 we give a brief description of the Pac-man game. In the next section we present our Pac-man Editor with special attention to the strategies of the ghost characters. Section 5 addresses related work and section 6 completes the exposition with a conclusion and a look at other implemented languages.

2 The DEViL System

The DEViL framework generates syntax-directed graphical structure editors for visual languages. The generated environments support all features of commonly used editors. Especially 2.5D views on the underlying semantic model are supported. A more in depth look at the generator framework and its generated products with respect to usability can be found in [12].

DEViL has already been successfully used for projects with nameable companies like Bosch [3], VW or SagemOrga [14]. The specification of this Pac-man Game Editor was one of many bachelor resp. master-theses that used the DEViL framework.

The specification process to generate "static" environments - environments without simulation and animation support - is divided into three parts. As can be seen later in this paper, simulation and animation support can be extended easily by the reuse of components of some of these three specification steps. Hence an user of the DEViL system who can build visual development environments can extend a language with simulation and animation support with reasonable effort.

To generate a structure editor for DEViL one first specifies the semantic model of the visual language. This is done with DSSL (DEViL Structure Specification Language). The semantic model abstracts from the visual representation. It stores just the information necessary to describe the semantics of the visual program. DSSL is inspired by an object oriented design with classes, inheritance, attributes and references. Fig. 1 shows a part of the specification of the semantic model for our Pac-man Editor. DEViL can generate an editor with a tree based structure manipulation view from this part of the specification.

```
CLASS Tile {
  columnRef: REF Column;
  item: SUB Item?;
}
ABSTRACT CLASS Item {
  name: VAL VLString;
}
CLASS Pacman INHERITS Item {
  direction: VAL VLInt INIT "2";
  angle: VAL VLInt INIT "0";
  clockwise: VAL VLBoolean;
}
```

Fig. 1. Part of the semantic model for the Pac-man Editor.

To obtain an advanced visual representation the semantic model (created with DSSL) is decorated with so called "visual patterns". Visual patterns define how constructs of the structure tree should look like. E.g. one can specify that some part of the structure tree should be laid out as the abstract concept "list" and aggregated nodes play the role of "list elements". Control attributes may modify this layout, for instance the list could be constituted vertically instead of horizontally. DEViL provides a huge library of precoined visual patterns with various possibilities to adapt their layout and appearance. A subset of this library is for example "sets, lists, trees, formulae or matrices". Technically, symbols of the semantic structure definition inherit from these visual patterns. The attribute evaluator generated by LIGA [4] of the underlying compiler generator framework Eli [5] computes the final graphical representation.

The last (optional) step of the specification process is the definition of a code generator. Here all of the tools of the Eli system to analyze the visual language instance can be used. A more detailed description of the VL specification process can be found in [13].

In order to separate concerns of specification simulation and animation are to be distinguished: simulation is the raw execution semantics of the visual language and animation is the smooth depiction of discrete execution of VL programs. Some visual languages have a precisely defined execution semantics, e.g. the firing of tokens in a Petri-net may be smoothly depicted by animation. For other visual languages simulation and animation may require to extend the semantic model to represent the simulation states or its graphical representation.

The presented Pac-man Editor (Fig. 4) considered as a visual language has a number of pawns that can be placed on a tile-based board which constitute the playing field. The pawns are typed structure objects of this VL. The Pac-man Editor has only four different pawns: "wall", "ghost", "powerpill" and "pac-man". Additionally, some structure objects are needed to represent the rows and columns of the board. Hence, our Pac-man VL is a playground editor where the user can create custom levels.

To specify a simulation for the Pac-man Editor we have to define the state space and the state transitions. Both can be specified in our simulation specification language DSIM.

Fig. 2 (a) shows the specification of the simulation model in DSIM. As can be seen, we again reuse DSSL concepts and we can extend the semantic model of the visual language to reach a new model that is suited for simulation. In this case we extended the semantic model class `Tile` (see Fig. 1). We can introduce new attributes that are needed for simulation purposes only or extend our simulation model with so called path expressions to traverse the simulation model tree at run time. Both model the state space for the simulation.

We could also narrow the semantic model of the visual language in our simulation model. This can be done if parts of the semantic model of the visual language are only needed for representation purposes and not for simulation.

```
MODEL {
  CLASS Tile {
    OBJECT pill OF PowerPill: "THIS.item.CHILDREN[0]";
    position: VAL VLPoint?;
    diffVal: VAL VLInt INIT "0";
    visited: VAL VLBoolean INIT "0";
  }
  CLASS Pacman {
    OBJECT tile OF Tile: "THIS.PARENT.PARENT";
  }
}
```

(a) Simulation model.

```
EVENTS {
  goGhost(Tile from, Tile to){
    Item i = REMOVE(from.item, FIRST);
    INSERT(to.item, i, FIRST);
  }
  eatPacman(Tile from, Tile to){
    IF(#[0]Root.sound == VLBoolean(1)){
      vlPlaySound("pacmanDeath.wav");
    }
    REMOVE(to.item, FIRST);
    Item i = REMOVE(from.item, FIRST);
    INSERT(to.item, i, FIRST);
    FIRE gameLost(#[0]Root)@TIME_NOW + 1;
  }
}
```

(b) Events.

Fig. 2. Simulation model in DSIM and some events which can be scheduled in the simulation loop.

```
FOREACH ghost IN [Ghost] {
  IF(ghost.strategy == VLInt(1)) {
    Tile to = NEIGHBOUR_TILE(mapping, NEUMANN, ghost.tile, Pacman);
    IF(NOTNULL(to) AND (ghost.eatable == VLBoolean(0))){
      FIRE eatPacman(ghost.tile, to) @TIME_DIRECT;
    }
    ELSE {
      IF(NOTNULL(#[0]Pacman)) {
        Tile to = NEIGHBOUR_EMPTY_TILE_RANDOM(mapping, NEUMANN, ghost.tile);
        IF(NOTNULL(to)) {
          FIRE goGhost(ghost.tile, to) @TIME_DIRECT;
        }
      }
    }
  }
}
```

Fig. 3. Part of the simulation loop.

Fig. 3 shows an excerpt of the behavior specification part of DSIM. Here the simulation model can be inspected and events can be scheduled that modify an instance of the simulation model. Hence we follow the event based approach to simulation. Events are scheduled for an arbitrary time. Any event can trigger arbitrary so called simulation modification actions. These actions modify the simulation model and they constitute the interface to the animation framework. The excerpt shows the behavior specification of the ghost pawns. They try to eat Pac-man if it is located on a neighbour tile. If not the ghost moves according to its strategy to the next tile.

In DSIM the following simulation modification actions exist which also form the interface to the animation part:

– `REMOVE` a structure object.

– `INSERT` a new structure object instance or insert a structure object, that was removed before. The latter would yield a `MOVE` action.
– `COPY` a structure object.
– `CHANGE_VAL` to change a primitive attribute or a reference.

In Fig. 2 (b) some events with corresponding simulation modification actions can be seen. The specific characteristics of the simulation modification actions is that an animation can automatically derived from such a specification.

The default animations triggered are: *slow shrinking* to invisibility of an object that is removed, *slow growing* of an object that is newly inserted. *Linear moving* (with optional easing) of a structurally moved object. Copied objects move from their copy source to their destination while changing their transparency value from invisible to visible. Since editors generated by DEViL are syntax directed structure editors, the creation or removal of structure objects can have side effects to other structure objects with respect to their size or position. These objects are automatically adapted smoothly, they are *morphed*. Even colors of structure objects are adapted smoothly.

The default animation behaviour is sufficient for most automatically derived animations as can be seen later. But, in some cases the default animation that is automatically triggered is not what the animator of a visual language desires. Here the animator can override the default behavior with so called animated visual patterns (AVPs). The AVPs can be decorated like the visual patterns to a structure object and tell the structure object in what way it is animated if a certain simulation modification action occurs. For instance if a token in a Petri-net is removed it should not shrink to invisibility which is the standard animation. The desired animation is to move the token to the fired transition, hence the used AVP to override the default behavior for remove is *AVPOnRemoveMove*. We have AVPs for changing size and transparency values of structure objects, for moving, scaling, rotating and so on. All of them can be combined and adapted to the needs of the animation.

A more detailed description of DEViL's simulation and animation facilities can be found in [2].

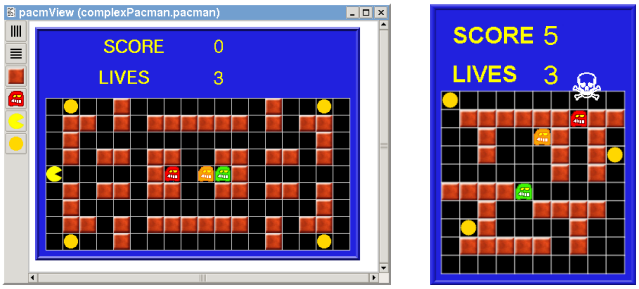
3 Pac-man

Pac-man is the most popular arcade computer game in the eighties of the last century and it was originally developed by Toru Iwatani for the Namco company in 1980. Because of the large degree of esteem different versions of the game have been reprogrammed many times for several systems like home computers, game-consoles and recently even for the iPhone [7]. The game is very interesting in terms of navigating a character around a structured playground, accumulating points, avoiding and (in some cases) attacking non-player game characters.

The classic version of Pac-man is an one-player game where the human player routes the Pac-man around a maze with the goal to avoid the four *ghost* characters and to eat as much pills as possible. Initially the pills are placed in each walk-in field of the playground and will be eaten via the achievement of the field by Pac-man. The overall four ghosts roam through the maze trying to catch Pac-man. This is successful when a ghost achieves a tile in which Pac-man is located.

In this case Pac-man looses one of his initial three lives and the game restarts when Pac-man has just one life. Each of the four ghosts pursues a different strategy to eat the Pac-man.

Besides the normal pills in each tile there are four *powerpills* which are located near each corner of a maze. When Pac-man eats a powerpill he gets a special score and is able to eat ghosts on his part. In this case all ghosts change their color to blue for few moments, reverse their direction, and usually move more slowly. If Pac-man eats a ghost, he gets a special score and the ghost resurrects in the middle of the maze after a few moments. In addition to the previously seen options there is one more possibility to increment the score: sometimes a symbol of a fruit appears at a random position of the maze, which also gives the chance to get extra points.



(a) Starting position of a Pac-man game.

(b) A dynamic animation object.

Fig. 4. Screenshots of our Pac-man game.

The game ends when all pills have been eaten or Pac-man has lost all of his lives. In the former case the player reaches a new level which is more difficult than the previous one. This can be achieved for example by faster moving ghosts.

Fig. 4 (a) shows a playground of a Pac-man game, which has been build with our Pac-man Editor. Besides the Pac-man the figure shows three different ghosts, wall items and powerpills.

4 Pac-man Editor

Our Pac-man Editor is structured as a *multi document interface* (MDI) and offers the ability to create user-defined playgrounds for Pac-man games. The user has the option to insert different items to the playground, e.g. Pac-man, ghosts, powerpills or wall items. It is also possible to expand the playground by adding rows and columns. A playground which is constructed in such a way allows to play Pac-man as mentioned above.

The specification of the semantic model was the first task to implement this editor. The most important part of the semantic model is the matrix structure. An object of the matrix class is associated with an arbitrary number of columns and rows. Each row owns several tiles, which includes in turn an item or not. The item is an abstract class and the concrete subclasses are either Wall-item, Pac-man, Ghost or Powerpill.

To realize a correct semantic playground it is essential to avoid more than one Pac-man or a game without Pac-man. Hence, the DEViL System provides

the ability to specify consistency constraints on various levels. E.g. cardinalities in the semantic model or specialized callback functions which can navigate the structure tree. All these checks are automatically performed before simulation. Hence, only a correct Pac-man game instance can be simulated.

Besides the consistency constraints the language designer can implement initialization functions for each class of the semantic structure. Such a function is a callback function and will be automatically called by the system, if a new object has been created. We used this for example to realize a default playground dimension of 10×10 tiles.

4.1 Strategies

With our editor it is possible to allot one of overall three different strategies to each ghost. We draw our inspiration with respect to the strategies of Repenning [11]:

Random The ghost roams randomly through the maze. At each step it evaluates the walk-in fields in the *von Neumann neighbourhood* and chooses one randomly.

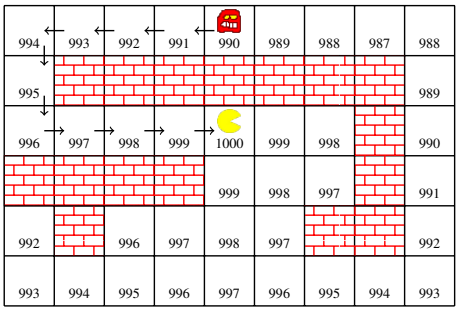


Fig. 5. Distribution of diffusion values to apply hill-climbing.

Incremental Approach The ghost tries to move closer to the Pac-man. At each step it evaluates the empty neighbour tiles and selects the closer closest one in euclidean sense.

Hill-climbing Due to the fact that the incremental approach does not permit the overcoming of walls, the strategy of hill-climbing affords this. To achieve this goal, diffusion values are used for each tile. These are used to spread the "scent" of the Pac-man in the maze. The value represents the closeness of a ghost to Pac-man. The largest value gets the tile in which Pac-man is allocated. Starting from this tile, the value is distributed to all walk-in fields of the playground. Every tile which is not accessible, e.g. a tile with a wall item, gets a negative diffusion value. At each step of the game the ghost selects the tile which has the largest diffusion value. Due to the fact that the diffusion values must be recalculated at each step, this brings the ghost closer to Pac-man. Fig. 5 illustrates the allocation of diffusion values and the way a ghost must go to get Pac-man.

A closer look to the implementation of the hill-climbing strategy is available in the next section. Amongst other things we describe the implementation of ghost strategies in DSIM.

4.2 Simulation

The user interaction via keyboard is essential for the Pac-man game. The DEViL System provides the ability to define arbitrary keyboard events which can be processed in the simulation.

We used the simulation model to add particular attributes which are necessarily needed for the simulation. An extract is given in Fig. 2 (a). We extend the Pac-man class with a tile attribute, which allows the access of the tile in which Pac-man is located, from the context of a Pac-man object. Besides others, we had extended the tile class with an attribute which stores the diffusion value of a tile. This is needed to realize the hill-climbing strategy. Keep in mind, that these attributes only exist in the simulation model, not in the semantic model of the Pac-man VL.

```
coordinatePacman(Pacman pacman, VLInt direction){
  Tile go = NEIGHBOUR_TILE(default, NEUMANN, pacman.tile, PowerPill);
  IF(NOTNULL(go)){
    FIRE eatPowerpill(pacman.tile, go, pacman, direction) @TIME_DIRECT;
  } ELSE {
    go = NEIGHBOUR_TILE(default, NEUMANN, pacman.tile, Ghost);
    IF(NOTNULL(go)){
      FIRE eatGhost(pacman.tile, go, pacman, direction) @TIME_DIRECT;
    }ELSE{
      go = NEIGHBOUR_TILE(default, pacman.tile, direction);
      IF(NOTNULL(go) AND (SIZE(go.item) == VLInt(0))){
        FIRE goPacman(pacman.tile, go, pacman, direction) @TIME_DIRECT;
      }
    }
  }
}

goPacman(Tile from, Tile to, Pacman p, VLInt d){
  FIRE incrementScore(#[0]Score, 1) @ TIME_DIRECT;
  FIRE computeRotation(p,d) @ TIME_DIRECT;
  Item i = REMOVE(from.item, FIRST);
  INSERT(to.item, i, FIRST);
}
```

Fig. 6. Coordination of the Pac-man pawn.

In the event block we specified events, which can be scheduled at an arbitrary simulation time in the loop block. Hence, our simulator follows an event driven approach. We had implemented overall 16 different events. Fig. 6 shows two events. The `coordinatePacman` event gets the Pac-man instance and a direction to move to. It checks, whether a powerpill or a ghost is in the way. If so Pac-man tries to eat the ghost resp. the powerpill. If there is nothing to eat Pac-man just walks to the next tile, the `goPacman` event is called. This event again calls two events to increment the score and to compute the rotation, which is needed for the animation. Finally the Pac-man pawn is removed from the actual tile and inserted to the tile in the desired direction. This yields a `MOVE` action for the Pac-man pawn.

```
NEIGHBOUR_COUNT(mapping, MOORE, pacman.tile, Ghost);
NEIGHBOUR_TILE(mapping, ghost.tile, S);
NEIGHBOUR_TILE_RANDOM(mapping, NEUMANN, pacman.tile);
```

Fig. 7. Exemplary neighbour access functions.

In each simulation step we have to compute the diffusion value for the hill-climbing strategy. This is done by a call of a C function. The function computes the value via a simple breadth-first search. Afterwards the ghost has to pick the target tile which has the largest diffusion value. To get a specific neighbour, we extended the simulation language such that we can access structure objects (of a specific type) in the neighbourhood of a given tile. Due to the fact that all editors generated with DEViL that make use of tiling have the same underlying

model we could identify a subset of tile-access functions which are often needed and generalize these functions. This lead to a decrease of hand written C-code.

Fig. 7 shows some neighbour access functions. The first function counts the ghosts in *Moore neighbourhood* of the Pac-man. A computation of the neighbour tile in south direction of a ghost shows the second function. The last function returns a random tile in *von Neumann* neighbourhood of Pac-man.

4.3 Animation

The default animation which is automatically derived from the simulation specification is almost adequate. A ghost and the Pac-man move fast from the start tile to the target tile in each simulation step. This is the case, because the animation framework interprets the modification actions `REMOVE` and `INSERT` as a *moving* animation. Furthermore the Pac-man shrinks to invisibility when he is caught by a ghost.

But Pac-man looks in the desired viewing direction until he has accessed the target tile. It would be nicer if Pac-man rotates to the desired viewing direction in the start tile before he moves to the target tile. Now the idea is to override the default behaviour for Pac-man. All animations are typed over their simulation modification action. Hence, we need to override the default animation pattern `MOVE`, because the Pac-man is moved (`REMOVE`d and `INSERT`ed) on the playground. We do it with the specification in Fig. 8 (a). We use the animated visual patterns `OnMoveRotate` and `OnMoveMove`. `OnMoveRotate` rotates a structure object if it is moved. Hence, we have to override the angle and rotate attributes. The angle and rotate attributes are stored in the "pacman" class (see Fig. 1) and will be computed via the *computeRotation* event in each simulation step. In addition we override the duration attribute to specify the duration of the rotate operation. In this configuration the rotation and the move are scheduled at the same simulation time, but we want the animation of the rotation to appear before the animation of the move. Hence the `OnMoveMove` animation must be animated after the `OnMoveRotate` animation. Hence, we have to assign the value 2 to the time attribute. Furthermore we override the duration attribute to indicate the duration time for a move operation. As can be seen, besides the simulation time, we have an animation time which defines an order of the animations and which can easily be adapted to gain a desired animation.

```
SYMBOL pacmView_Pacman INHERITS VPContainerElement, VPForm,
       AVPOnMoveRotate, AVPOnMoveMove, AVPOnRemoveShrink
COMPUTE
  SYNT.drawing = ADDROF(PacmanDrawing);
  SYNT.onMoveRotateAngle = THIS.pers_angle;
  SYNT.onMoveRotateClockwise = THIS.pers_clockwise;
  SYNT.onMoveRotateDuration = 600;
  SYNT.onMoveMoveRaiseDisplayOrder = 1;
  SYNT.onMoveMoveAnimationTime = 2;
  SYNT.onMoveMoveDuration = 900;
  SYNT.onRemoveShrinkAnimationTime = 10;
END;
```

(a) Mapping of AVPs with control attributes to override default animation.

```
SYMBOL pacmView_Pacman INHERITS AVPCreateDynamicObject,
    AVPMoveDynamicObject
COMPUTE
  SYNT.createDynamicObjectModificationAction = REMOVE;
  SYNT.createDynamicObjectDrawing = ADDROF(SkullDrawing);
  SYNT.createDynamicObjectPosition = POSITION(
          SELECT(THIS.oPosition, sub(VLPoint(0,10))));

  SYNT.moveDynamicObjectDuration = 8000;
  SYNT.moveDynamicObjectStartPosition = POSITION(
          SELECT(THIS.oPosition, sub(VLPoint(0,10))));
  SYNT.moveDynamicObjectEndPosition = POSITION(
          SELECT(THIS.oPosition, sub(VLPoint(0,60))));
END;
```

(b) Creating a dynamic animation object and moving it.

Fig. 8. Animation of Pac-man.

The animation framework offers the possibility to animate objects which are not part of the semantic model (so called *dynamic objects*). If Pac-man is caught,

we have used this feature to display a skull (see Fig. 4 (b)). For such a purpose it is only necessary to use the provided visual patterns as described in Fig. 8 (b). The visual pattern `CreateDynamicObject` reacts to a modification action and offers the possibility to add a drawing. As seen in Fig. 8 (b) we override the modification action attribute to react to a remove action. Furthermore, we override the drawing attribute to add the skull drawing. In order that the skull moves bottom-up from the position of the Pac-man, we had used the pattern `MoveDynamicObject`. We also had used the pattern `OnRemoveShrink` to show the skull temporary.

The specification of an animation in DEViL is straight forward: first specify a simulation, then derive the animation automatically. Hence the animation is a formal mapping of its simulation part. At last animations can be adapted by overriding the default animations through the application of a huge declarative animation pattern library.

5 Related Work

The Agentsheets system [10] can generate tile based simulations and games. The specification process is fully graphical and rule based. Agentsheets uses the programming by demonstration paradigm. In the rules one can access neighbour tiles through the help of icons with specific arrows. This is the visual variant of our neighbour access functions. Agentsheets is restricted to tile based simulation whereas our system can also handle diagrammatic or iconic visualizations.

In the area of generator frameworks for visual language environments the GenGed [1] system makes use of graph transformation and visual rewrite rules to specify simulation. To store the simulation state, rules must be extended. This is similar to our simulation model which can extend the semantic model of a VL. GenGed uses a formal mapping between simulation and animation. This is comparable to our simulation modification actions which trigger default animations. They are the interface between simulation and animation framework. Smooth and complex animations can not be specified with GenGed.

The DiaGen [6] system also uses graph transformation to specify a visual language. Some editors already support simulation and animation. Interesting is that every animation step is a state of the system whereas we interpolate between two adjacent simulation model states.

6 Conclusion

The specification of the Pac-man editor is a straight forward task. Table 1 shows that we needed 220 LOC for the whole simulation part including all ghost strategies and 400 LOC for hand written C-code. The other specs part needs only 236 LOC. The second column of the table shows a decrease of total LOC from 883 to 646 LOC. This is because of the neighbour access functions we had generalized. This reduced the LOC of C-code nearly by 250 LOC and we need only 7 additional LOC in the simulation specification to realize to mapping between concrete and generalize matrix structure. The 156 LOC of C-code is just a simple tile initialization function for the hill-climbing strategy. The automatically

derived animation is sufficient to play the game. The 27 LOC are just syntactic sugar.

The DSIM language with its narrow interface to the animation and its constructs tailored for the simulation of visual languages has already been approved in other VLs with execution semantics like Petri-nets, a Datapath simulation, electronic circuits and even the game Ludo. Table 2 shows the amount of LOC for simulation and animation part of already implemented editors. The examples in line two and three are based on the Petri-net simulation shown in line one. They show the simulation of the well-known dining philosophers and a simulation of a signal light of a four-way-crossing. Both are structural coupled to the Petri-net with DEViL's internal declarative coupling mechanism. A simulation of the Petri-net automatically triggers synchronization functions in the philosophers resp. signal-light view. The simulator detects these triggerings and calls the animation framework. Hence, additional specification amount is not needed.

	LOC	**LOC with access fct.**	**generated LOC**
simulation	220	227	87.504
animation	27	27	
C-code	400	156	
other specs.	236	236	
	883	646	87.504

Table 1. Distribution of the specification complexity.

	Simulation	**Coupling**	**Animation**	**Anim. syntactic sugar**
Petri-nets	29		4	0
Dining-Philosophers	29	95	4	0
Signal-Lights	29	57	4	0
Logo	211		3	3
Game of Life	39		0	0
Ludo	338		0	0
Statecharts	78		2	0
Bubblesort	13		0	0
Quicksort	93		0	0
CPU Datapath	263		160	0
Washing bay	35		0	0
Electronic circuits	99		109	0

Table 2. Simulation and Animation LOC of other VLs.

As can be seen in Table 2 the automatically triggered animation is mostly sufficient. We need to adapt the animation only in simulations where animations depend on the context of their structure objects. E.g. this is the case in our CPU datapath simulation where an animation of an instruction is different whether it is located in an instruction decoder, in an accumulator or somewhere else.

The already implemented VLs have a very diverse appearance: we have diagrammatic, iconic and graph based depictions. Currently we are working on a traffic simulation to see how our approach scales. Interesting extensions would be semantic zooming, camera views or even isometric views. Here also a pattern based approach is imaginable.

Also outstanding is a visual language for DSIM and an usability study.

References

1. Bardohl, R.: GenGed: A generic graphical editor for visual languages based on algebraic graph grammars. In: 1998 IEEE Symp. on Visual Lang. pp. 48–55 (Sep 1998)
2. Cramer, B., Kastens, U.: Animation automatically generated from simulation specifications. In: VLHCC '09: Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 157–164. IEEE Computer Society, Washington, DC, USA (2009)
3. Cramer, B., Klassen, D., Kastens, U.: Entwicklung und Evaluierung einer domnenspezifischen Sprache fr SPS-Schrittketten. In: Fahland, D., Sadilek, D.A., Scheidgen, M., Weileder, S. (eds.) DSML. CEUR Workshop Proceedings, vol. 324, pp. 59–73. CEUR-WS.org (2008), http://dblp.uni-trier.de/db/conf/dsml/dsml2008.html#CramerKK08
4. Kastens, U.: An attribute grammar system in a compiler construction environment. In: Proceedings of the International Summer School on Attribute Grammars, Application and Systems. Lecture Notes in Computer Science, vol. 545, pp. 380–400. Springer Verlag (1991)
5. Kastens, U., Pfahler, P., Jung, M.: The Eli system. In: Koskimies, K. (ed.) Proceedings of 7th International Conference on Compiler Construction CC'98. pp. 294–297. No. 1383 in Lecture Notes in Computer Science, Springer Verlag (Mar 1998)
6. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (Aug 2002), http://www.elsevier.com/gej-ng/10/39/21/86/49/29/abstract.html
7. Namco Games: Pacman for iPhone. http://www.appsafari.com/games/2741/pacman-for-iphone/ (2008), [Online; accessed 19-February-2010]
8. Norman, D.A., Draper, S.W.: User Centered System Design; New Perspectives on Human-Computer Interaction. L. Erlbaum Associates Inc., Hillsdale, NJ, USA (1986)
9. Object Management Group: Unified Modeling Language (UML), version 2.2 (2009), http://www.omg.org/technology/documents/formal/uml.htm
10. Repenning, A.: AgentSheets®: an Interactive Simulation Environment with End-User Programmable Agents. In: Interaction 2000, Tokyo, Japan (2000)
11. Repenning, A.: Collaborative Diffusion: Programming Antiobjects. In: OOPSLA 2006, ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, (Portland, Oregon, 2006). IEEE Press (2006)
12. Schmidt, C., Cramer, B., Kastens, U.: Usability evaluation of a system for implementation of visual languages. In: Symposium on Visual Languages and Human-Centric Computing. pp. 231–238. IEEE Computer Society Press, Coeur d'Alne, Idaho, USA (Sep 2007)
13. Schmidt, C., Kastens, U., Cramer, B.: Using DEViL for implementation of domain-specific visual languages. In: Proceedings of the 1st Workshop on Domain-Specific Program Development. Nantes, France (Jul 2006), http://ag-kastens.upb.de/paper/dspd2006-devil.pdf
14. Schmidt, C., Pfahler, P., Kastens, U., Fischer, C., Gmbh, O.K.: Simtelligence designer/j: A visual language to specify sim toolkit applications. In: Proceedings of the Second Workshop on Domain Specific Visual Languages (OOPSLA 2002 (2002), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.9269