Animation automatically generated from simulation specifications

Bastian Cramer and Uwe Kastens
University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany
{bcramer, uwe}@uni-paderborn.de

Abstract

Our generator framework DEViL supports the development of visual languages. It generates complete language implementations from high-level specifications including advanced graphical structure editors. It has been successfully used for a wide range of domain specific visual languages. For a DSL that has an execution semantics, e.g. a processor specification language, it is desirable to simulate and to visualize program execution for purposes of analysis or evaluation.

This paper shows how DEViL is extended to generate a simulator for a visual language from specification of its state transition model. Without the need of any further specification a smooth animation of program execution using a technique of graphical interpolation is generated automatically. Further advanced animations can easily be obtained by simply associating some "animated visual patterns" to standardized operations of the simulator. DEViL provides a large variety of such patterns which encapsulate the implementation of certain useful animation effects. Our approach has proven to be effective for the animation of several DSLs.

1. Introduction

Today visual languages (VLs) are often used in the software engineering process, UML is a well-known and wide spread example. Visual languages can use graphical metaphors of a specialized domain. Due to the use of graphical editors the specification process is on a very high abstraction level and expert knowledge can be incorporated into the editor. This allows even non-programmers to develop software.

Toolsets like DEViL (Development Environment for Visual Languages) can be used to generate such graphical structure editors from high-level specifications. Hence, it is possible to derive even advanced editors with reasonable little effort and use it in the rapid development process. Indeed, todays software design process bases on static diagrams. The simulation and animation of such a static visual language would be the next step. It could bridge the gap between program and program execution which must always be in the programmers mind. In particular animated representations can help to understand parallel processes. Even in imperative sequences animation support is far better than the raw textual representation. The simulation and animation of visual languages can also lead to a better understanding of a complex system and potential errors can be recognized in a very early stage.

In this paper we show how the visual language generator – DEViL – is extended such that it can generate editors for visual languages with simulation and animation support. We show that this is realized because of the reuse of existing and established concepts.

In the next section we will introduce DEViL and its underlying specification concepts. We show which steps are needed to generate an editor for a visual language.

In the third section we introduce the simulation framework and "DSIM" a simulation language tailored to the special needs of visual language simulation.

Section four introduces the main thesis of this paper: the automatic generation of animation from a simulation specification.

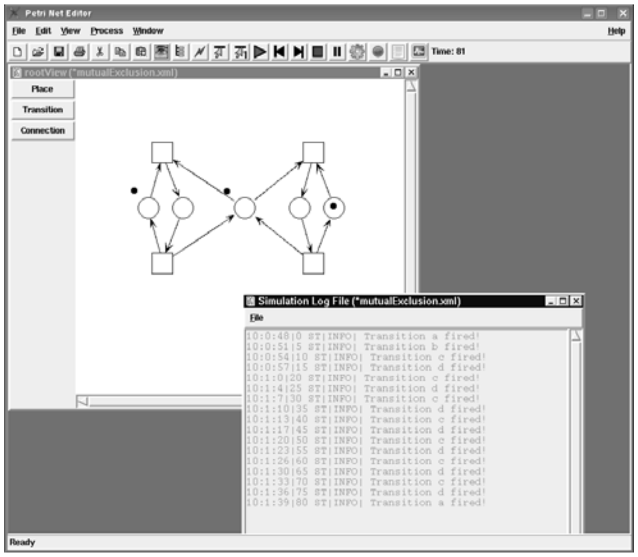
Finally we discuss related work and conclude this paper.

2. DEViL

The toolset DEViL can generate syntax-directed graphical editors from high-level specifications. The generated editors offer a multi-document environment, i.e. 2.5d views to the model are possible and well-known features of actual editors like copy-and-paste, printing, save and load of examples as well as search and replace.

A more complete discussion of properties of generated products and their usability can be found in [16].

Furthermore DEViL supports many functions to do a full semantic analysis on the generated structure editors like car-


Petri Net Editor
File Edit View Process Window Help
Time: 81
rootView (*mutualExclusion.xml)
Place
Transition
Connection
Simulation Log File (*mutualExclusion.xml)
File
Ready

```
10:0:48|0 ST|INFO| Transition a fired!
10:0:51|5 ST|INFO| Transition b fired!
10:0:54|10 ST|INFO| Transition c fired!
10:0:57|15 ST|INFO| Transition d fired!
10:1:0|20 ST|INFO| Transition c fired!
10:1:4|25 ST|INFO| Transition d fired!
10:1:7|30 ST|INFO| Transition c fired!
10:1:10|35 ST|INFO| Transition d fired!
10:1:13|40 ST|INFO| Transition c fired!
10:1:17|45 ST|INFO| Transition d fired!
10:1:20|50 ST|INFO| Transition c fired!
10:1:23|55 ST|INFO| Transition d fired!
10:1:26|60 ST|INFO| Transition c fired!
10:1:30|65 ST|INFO| Transition d fired!
10:1:33|70 ST|INFO| Transition c fired!
10:1:36|75 ST|INFO| Transition d fired!
10:1:39|80 ST|INFO| Transition a fired!
```

Figure 1. A generated editor simulating a Petri-net

```
CLASS Root {
  nodes: SUB Node*;
  connections: SUB Connection*;
  setSize: VAL VLPoint? INIT "300 300";
}
ABSTRACT CLASS Node {
  position: VAL VLPoint? EDITWITH "None";
  name: VAL VLString;
}
CLASS Place INHERITS Node {
  marks: VAL VLInt INIT "0";
}
CLASS Transition INHERITS Node {

}
CLASS Connection {
  from: REF Node EDITWITH "None";
  to: REF Node EDITWITH "None";
  position: VAL VLPoint? EDITWITH "None";
  weight: VAL VLInt INIT "1";
}
```

Figure 2. Specification of the abstract structure of a Petri-Net editor

dinalities, check functions and the integration of user defined code. Internally DEViL uses the compiler generator framework Eli [8], hence all of Eli´s features can be used as well, especially the code generators to do a source-to-source compilation from the visual language to an arbitrary target language.

DEViL has already been used to create visual languages in cooperations with nameable companies like Bosch [5], VW or SagemOrga [17] as well as in many bachelor or master theses.

To generate a structure editor with DEViL one first specifies the semantic model of the visual language. In DEViL this is done with DSSL (DEViL Structure Specification Language) which is a specialized language that is inspired by object-oriented concepts like classes, inheritance, aggregation and the definition of attributes. Attributes represent primitive values like text or references to other structure objects. Fig. 2 shows the specification of the semantic model for a Petri-net editor. A class "Root", which is the base class for structure editors in DEViL, defines the aggregation of nodes and connections. Nodes can either be of the concrete types "Place" or "Transition". With the specification of this abstract structure DEViL can generate a simple structure editor, which allows editing the structure in a tree based view. To achieve advanced graphical representations the so called *visual patterns* can be applied to the semantic model of the visual language (see figure 3, step 2a).

Visual patterns describe in which way structure trees should be represented graphically. They are an abstract concept which can be used in a declarative way. For instance one can specify that some part of the structure tree should be represented as the abstract concept *"list"* and some aggregated nodes of the list play the role of *list elements*. DEViL provides a library of common used visual patterns. These patterns can be used and combined in an arbitrary way. A subset of this library is for example *"sets, lists, formulae, matrices"* or *"tables"*. All patterns can be parameterized through attributes to reach a specific layout.

Technically visual patterns are applied to the structure tree (created with DSSL) with attributed grammars. The attribute evaluator generator LIGA [10] of the Eli system computes the final graphical representation.

To analyze and transform a visual language it is important to have specialized constructs to navigate through

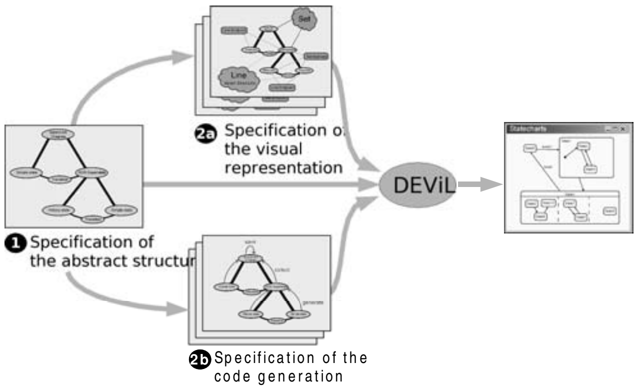

Set
Line
2a Specification of the visual representation
DEViL
1 Specification of the abstract structure
2b Specification of the code generation
Statecharts

Figure 3. Specification process in DEViL

the structure tree and read attributes of the structure tree. In DEViL we use so called *path expressions*. They allow an easy access to all parts of the structure tree and they are comparable with XPath expressions in XML documents [20].

The specification of the Petri-Net editor introduced above needs 39 lines of code (LOC) for structure specification and 65 LOC for view specification. The simulated and animated version needs 29 LOC for simulation specification and 4 LOC for the animation. Hence, a simulated and animated Petri-net editor can be derived with 137 LOC.

3. The simulation framework

Before we introduce the simulation framework we want to classify the term "simulation" and "simulation model". The simulation of a system is the execution of an abstract model (the simulation model) which describes an abstract concept or parts of the real world. In this model some sort of simulation entities flow through the model and alter its state. The simulation execution results in an amount of data which is analyzed in several ways. It could be visualized as graphs or in an animation framework which shows the stepwise change in the model.

We regard the simulation of a visual language as a set of transformations of the semantic model of the visual language. At this point we abstract from the graphical representation, e.g. an animation or a specific layout is not considered yet. The simulation describes the raw execution semantic of a visual language.

An important aspect with respect to the simulation is the simulation model. It is the foundation of the simulation. At runtime instances of the model store states of the simulation. The simulation model must be suited to the needs of the simulation, e.g. it must be as small as possible to prevent long running simulations. Hence, in the simulation model only constructs appear that are necessarily needed for the simulation. But on the other hand it must be as flexible to store information that is only needed by the simulation.

The simulation model of a visual language is not always equal to the semantic model of the VL. It can hold some extra constructs, but also fewer constructs are possible, if the VL has a huge graphical extend which is not needed by the simulation. There are many approaches to construct a model (declarative, functional, spatial, multi-model...[7]). All models have in common that there are some sort of simulation objects that move in the model and change its internal state.

This can be adapted to visual languages as well. Simulation objects are part of the visual language. They are created or removed in the model or their attributes are modified. This leads to the fact that a simulation language for visual languages needs specialized constructs to walk through the simulation model and modify it.

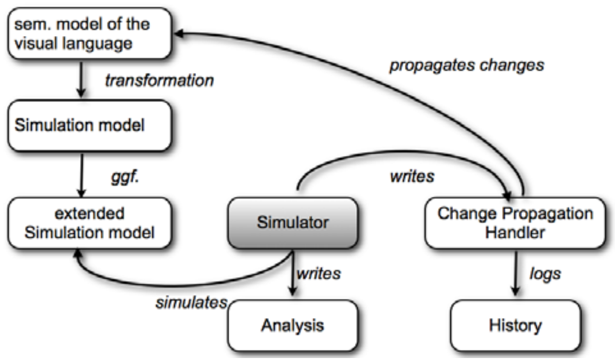


Figure 4. The simulator concept

To develop a simulator and a simulation specification language, we analysed common simulation languages and libraries. On the simulation language side we analysed amongst others Siman [13] and GPSS [18] and simulation libraries like CSIM [19]. They allow the discrete event based simulation which is frequently used in simulation, because of its simplicity and expressiveness power.

All simulation languages define simulation runs over an abstract simulation time which can be treated very flexible. Not only the compression of time is possible, but also shifting back to earlier simulation steps. The event oriented approach with a priority based waiting queue for events is most often used. To achieve significant simulation results for analysis, a simulation model with random numbers of different varieties and the tracing or logging of variables is needed. Most of the simulation languages support these features.

The studied simulation languages and libraries all have in common that simulation objects are created and manipulated while they run through the simulation model. Some simulation languages allow the grouping of and the easy access to simulation objects. This is important for visual languages, because often we want to access visual language constructs with specific properties e.g. "all tokens in the precondition of a place" in a Petri-net.

3.1 The simulator

The structure of the generated simulator framework looks like in figure 4.

The simulator simulates on its own simulation model which might be extended and bases on the semantic model of the visual language. If a simulation is build on top of an existing visual language one realizes that the semantic structure of the visual language does not fit to the intended simulation structure. It contains classes, attributes or references that are an admission to the needs of the graphical representation. Hence there are some sort of objects of the

VL that are needed for the simulation and some that are not. This results in a simulation model that can fully map the semantic model of the visual language, but can also reduce it to the simulation needs. The extension of the model is needed to satisfy the requirements of random numbers, waiting queues and path expressions of the simulation. The decoupling of the simulator from the semantic model of the VL has the advantage that existing specifications of structure editors (there are many in DEViL) should not be influenced so that no side effects occur.

If there is no simulation definition for a specification the simulator should not be integrated in the editor. The simulator should have the ability to stand alone. This separation means that the simulator can simulate on its own without underlying parts of the visual language model.

If a visual language with simulation extension is developed in a team it is possible that parts of the specification needed by the simulation are not yet integrated in the visual language model. Hence the simulator needs virtual language constructs that can be transferred to the visual language later on.

The simulator writes changes into its own simulation model and tells the so called change-propagation-handler (CPH) the differences to the visual language model. The CPH writes the changes back to the semantic visual model if needed. The propagation interval can be set by the user. E.g. the user can say, tell me the simulation results after *n* simulation steps. The default propagation interval is *1*, hence every simulation step is transferred directly in the VL. Furthermore the CPH writes the history, which allows the stepwise undo of simulation steps.

The simulator writes simulation results in an analysis module. This module logs simulation steps and generates workload graphs of queues or random variables. Another important aspect of the simulator is, that it works event-based and follows the "next-event-time approach" in which the event queue is advanced to the next event instead of incrementing the simulation time by one, each step.

3.2 The language DSIM

Our simulation specification language DSIM is used to specify the simulation of visual languages. As mentioned in the previous section DSIM needs constructs to retrieve objects of the semantic model that are relevant for the simulation and integrate them into the simulation model. Beside the simulation model we need language constructs to express the real simulation e.g. the execution semantics of our simulation and finally we need the declaration of discrete events to specify the modification of instances of the simulation model.

Hence, DSIM consists of three parts: the *simulation model* definition, the *conditions* block and the *event* definition block.

In the simulation model part of the language we can modify and extend the semantic visual model which results in our simulation model. The extension is achieved through the definition of extra simulation classes and simulation specific attributes like priority queues, sets and random variables. Hence, the simulation model for the visual language is a 3-tupel $\mathcal{S} = (\mathcal{C}_{\text{sim}}, \mathcal{A}, \mathcal{M})$, where $\mathcal{C}_{\text{sim}} = \mathcal{C}_{\text{semantic}} \cup \mathcal{C}_{\text{ext}}$ is the set of classes of the visual semantic model unified with the set of extended simulation classes.

All classes of the semantic visual model can, but do not have to occur in the simulation model. If a class is taken over from the semantic model of the visual language to the simulation model, it is the complete clone of its VL model pendant.

$\mathcal{A} = \mathcal{P} \cup \mathcal{V} \cup \mathcal{Q}$ is the set of attributes, consisting of path expressions, primitive and queue attributes and random variables.

$\mathcal{M} : \mathcal{C}_{\text{sim}} \times \mathcal{A}$ is a mapping of attributes to classes.

The simulation model introduces new attributes which decorate the classes during a simulation run. The most often used attributes in the simulation model are *sets*. They define a collection of structure objects which satisfy a specific condition. The condition itself is specified by a path expression. In an object oriented sense, sets specify methods of classes. An example of a simulation specification for Petri-Nets can be seen in figure 5. The simulation model in the upper part introduces the simulation class `''Transition''` which is extended by two sets called `''incomingPlaces''` and `''outgoingPlaces''` that compute the pre- resp. post-condition of a specific transition and both hold objects of type *"Place"*.

The conditions block specifies the real simulation. It is looped by the simulator. Here the simulation model can be accessed and events can be scheduled. The access of the model is realized through special language constructs like `FOREACH`, `IFSOME`, `IFEVERY`,... which satisfy the requirement of easy access of simulation languages.

In the event block events are specified which can be scheduled in the conditions part. Events modify the instance of the simulation model. Remarkable is that DSIM offers modification on single simulation objects as well as on sets of objects. This leads to an easy specification process.

In the conditions part of the simulation specification the pre-condition of every transition is checked. If all places of the pre-condition have at least one token, the transition can fire the `''transitionFire''` event which is defined in the events part of the specification. In it the event `''transitionFire''` is defined. It decrements the tokens of the pre-condition of the firing transition (`REMOVE`) and increments the token amount in the post-condition (`INSERT`). Please keep in mind, that a new(!) token is created in the post-condition.

```
SimulationModel{
 CLASS Transition {
  SET incomingPlaces OF Place:
    "THIS.IVALUE[Connection.to].
    PARENT.from.VALUE[Place]";
  SET outgoingPlaces OF Place:
    "THIS.IVALUE[Connection.from].
     PARENT.to.VALUE[Place]";
 }
}
```

Simulation-model

```
Conditions {
  FOREACH t IN Transition RANDOM{
    IFEVERY SIZE(t.incomingPlaces->tokens)
     > 0 {
        FIRE transitionFire(t) @ TIME_DIRECT;

    }
   }
}
```

Condition-block

```
Events{
  transitionFire(Transition transition) {
  FOREACH place IN transition.incomingPlaces {
   Token token = REMOVE(place->tokens);
  }
  FOREACH place IN transition.outgoingPlaces {
   INSERT(place->tokens, "Token");
  }
 }
}
```

Event-block

Figure 5. Simulation specification for Petri-Nets

The DSIM language also allows the specification of random variables and queues in the simulation model. They are used for the simulation of queue based systems like washing bays or assembly lines. They can be used like sets above:

```
CLASS Assembly {
 QUEUE washingBay
     OF Car: FIFO washingBay MAX 1;
 RANDOM normalVariety
     OF NORMAL: 5, 2;
}
```

Up to this point we can only *simulate* visual languages. That means, the discrete visualization of simulation steps. Now we want to get a smooth animation from our simulation specification.

4. The animation framework

A possible approach towards an animation could be the annotation of the simulation specification. This specification mechanism is inspired by the *"interesting events"* approach used widely in algorithm animation systems and first introduced in *BALSA "'Brown University Algorithm Simulator and Animator"'* [2]. So our first code example looks like

```
 ...
 FOREACH place IN incomingPlaces {
  ANIMATION_ACTION ``tokenMove''
      (place, transition);
 }
 ...
```

A so called animation action is placed at interesting points in the simulation specification, here somewhere inside a firing event. It gets a name and some actual parameters. The definition of the animation action looks like this:

```
ANIMATION_VIEW petriNet {
 tokenMove(Place p, Transtition t) {
  MOVE "tokenDrawing" (POSITION(p),
                       POSITION(t),
         slowInSlowOut, 2.5sec;
 }
}
```

The concrete implementation of the animation action with name ``tokenMove'' moves a drawing named ``tokenDrawing'' from the position of the formal parameter Place p to the Transition t. The move is parameterized with easing and has a duration of 2.5 seconds.

This imperative approach has many disadvantages:

- Size and position of language constructs after one simulation step can only be estimated. E.g. erasing of an object has consequences for other objects. Adjacent object have to move up.
- The animator has to rely on the simulator, because the simulator has to annotate his code with adequate calls to animation functions. This yields to a mixture of simulation and animation. Logic and layout are not separated.
- Through the use of animation function calls the coupling between simulation and animation is very loosely. Furthermore a mapping between both layers is not forced. This may lead to inconsistent animations that do not reflect the real simulation.

All these drawbacks lead to a new specification approach: the *linear graphical interpolation*.

4.1 Linear graphical interpolation

Since the graphical state of two adjacent simulation steps is well-known – it can be computed through two attribute

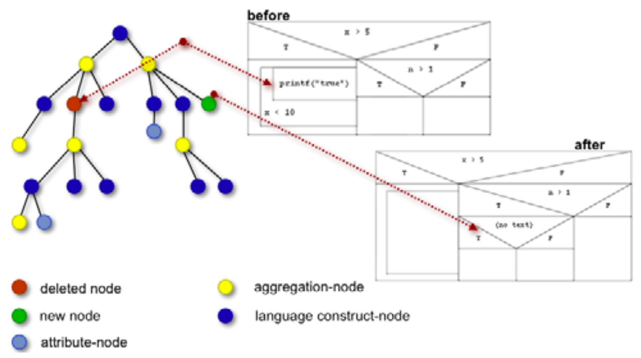


Figure 6. Graphical Interpolation

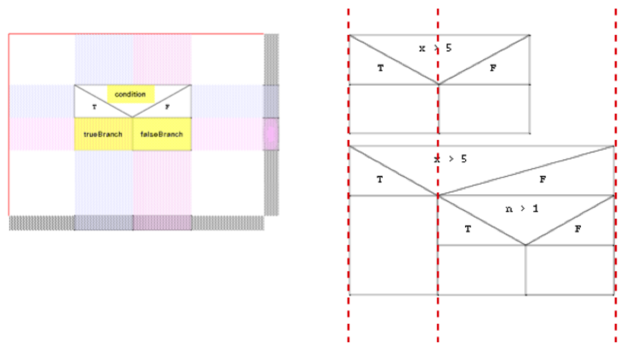


Figure 7. Generic drawing on the left and morphing on the right

evaluator runs – the size and position of all structure objects can be stored. The next step is to graphically interpolate between these two snapshots. They can be regarded as keyframes for the animation. Fig 6 shows a structure tree representing a simulation run and both graphical representations of a Nassi-Shneiderman diagram in which one node was deleted and another node that was created. The dashed arrows show the correspondence between structure nodes and their graphical representation.

This observation leads to our animation approach:

> "Linear graphical interpolation is the standard. Variations of it must be specified (declaratively)."

To understand this statement we have a closer look at the DSIM specification language: There are four kinds of simulation modification actions:

1. `CREATE`, is applied to an aggregation node and creates a new object of a given type.
2. `REMOVE`, is applied to an aggregation node and deletes the first object in the aggregation.
3. `MOVE`, is used to move complete structure trees.
4. `CLONE`, is used to make a deep copy of a structure object.
5. `CHANGE_VAL`, is used in primitive attribute nodes and indicates a change in value.

The `INSERT` modification is an abbreviation and is either of type `MOVE`, if a structure tree is inserted or of type `CREATE`, if the structure tree did not yet exist.

These four simulation modification operations trigger our animation framework and cause the execution of standard animation actions.

These are: the slow growing of an object to its final size if it is newly created. The slow shrinking of an object if it is deleted and the linear move of an object if it is moved structurally. A copied object is depicted by a move from the copy to the destination. The transparency value of the copied object is slowly adapted from invisible to visible while it is moving. The modification of a primitive attribute can cause a stimuli in the upper context of the corresponding node.

This linear interpolation creates smooth animation for most languages, but the *linear* interpolation is not sufficient. Structure objects do not grow linear if new objects are inserted.

Fig 7 shows the situation. The Nassi-Shneiderman diagram (fig. 7 upper right) expands not linearly if a new node is inserted (fig. 7 lower right) which is implied through the red dotted lines. This phenomenon results from the specification method of generic drawings in DEViL. They are used to get a concrete graphical representation and can hold containers to aggregate other nodes. To satisfy the space needs of all language constructs, containers (fig. 7 left, yellow rectangles) can grow independently which results in a space-saving result.

The *linear* graphical interpolation does not work here, but DEViL can automatically transform generic drawings, they are *morphed*. This leads to an smooth animation.

The default animation actions are useful in many animations, but sometimes an animator wants an animation that differs from the default animation. A look at the Petri-net simulation specification shows, that tokens are removed in the pre-condition and created (newly inserted) in the post-conditions. This would lead to an animation which shrinks tokens to invisible if they are removed and grows newly created tokens to their final size if they are created. Actually we want an animation where tokens fly through the Petri-net. Hence, we have to overwrite the default animation mappings. We do that with the so called *"animated visual patterns"* (AVP). The AVPs are

```
SYMBOL rootView_Token INHERITS VPForm,
 VPSetElement, AVPOnRemoveMove,
  AVPOnCreateMove
COMPUTE
  SYNT.drawing= ADDROF(TokenDrawing);
  SYNT.onRemoveMoveEndPosition    = POS(1);
  SYNT.onRemoveMoveAnimationTime = 1;
  SYNT.onCreateMoveStartPosition = POS(1);
  SYNT.onCreateMoveAnimationTime = 2;
END;
```

Figure 8. The mapping of AVPs with control attributes

comparable with the already introduced visual patterns, because they specify certain representation aspects in a declarative way and both are specified in LIDO.

To achieve our Petri-net animation we overwrite the `CREATE` and `REMOVE` default animation patterns. We do it with the specification in fig. 8. We use the animated visual patterns `OnRemoveMove` and `OnCreateMove`. `OnRemoveMove` moves a structure object to a certain position if it is removed. `OnCreateMove` moves a structure object from a certain position to its final position. Hence, we have to overwrite the position control attributes of both AVPs: `SYNT.onRemoveMoveEndPosition` resp. `SYNT.onCreateMoveStartPosition`. At last we overwrite the timing parameters. Both simulation actions `CREATE/REMOVE` are scheduled at the same simulation time, but we want the animation of a removed token to appear before the animation of a create token, because the token flies from its position in the pre-condition to the firing transition and then to its final position in the post-condition. Hence the `OnRemoveMove` animation must be animated before the `OnCreateMove` animation. We achieve this by overwriting the default animation control attributes `SYNT.onRemoveMoveEndPosition` resp. `SYNT.onCreateMoveAnimationTime`.

Altogether we implemented a library of animated visual patterns. Some of them are *Move, Blur, Blink, Flash, Rotate, Explode,* etc.. All of them are highly configurable through control attributes, e.g. every move can be parameterized to achieve easing or anticipation. Furthermore an arbitrary combination of the AVPs is possible.

The declarative AVP approach compensates the disadvantages of the earlier mentioned imperative specification style. Mainly it provides a very smooth animation which is pretty hard to achieve in syntax directed structure editors. The AVPs specify **what** should happen and not **how**. The mapping between simulation and animation is formally correct. Animations are not missed and the generator takes care that all simulation modifications are considered in the animation framework. The simulation and animation specifications are separated which supports team work and a separation of logic and layout.

5. Related work

In the area of simulation we studied well-known simulation languages like Simscript, GPSS or Siman. We adopted their concepts and integrated it into our simulator. In particular generators for domain specific languages that can simulate visual languages using random variables and priority queues are not known to us.

To simulate visual languages often Abstract State Machines [9] and the Abstract State Machine Language (AsmL) [11] by Microsoft are used. In [3] AsmL is used to specify the behavior of DSLs.

Another approach to simulation is the use of graph transformation and visual rewrite rules which is used in the GenGed System [1] and the extension of Ermel et al. [6]. Here the mappings between semantic model of the VL and the simulation resp. animation are described very formally in the way of rule-based graph transformations. This leads to a provable simulation and animation mapping. Smooth animation and complex animations can not be specified.

DSIM is tailored to the needs of visual languages and supports many constructs to walk in the structure tree and to access structure objects. It is not as general as ASMs, but the generated animation is also a very formal mapping from the simulation. Hence, errors of the animation can be reduced to a wrong simulation specification.

In [14] the Alma system is used for program- and algorithm animation. Every modification in the structure tree causes a visualization. This can be compared to the simulation modification actions in DSIM.

The Amulet framework [12] (C++ libraries) allows to annotate graphical objects of a GUI. The framework asks the objects state and triggers animation actions if changes are detected. A change in an integer variable from 0 to 100 can cause an animation that counts slowly from 0 to 100. Amulet distinguishes many states of an object and animation actions can be combined very flexible. The state changes can be compared to DSIMs for basic simulation modifications.

Pavane [4] is a program visualization tool that uses graphical interpolation to retrieve an animation.

ZVMT [15] is a library for the animation of graphical user interfaces. It uses a declarative animation approach where arbitrary animations can be attached to all sorts of graphical objects. It supports multiple camera views and the path-transition paradigm for camera views which could be a good extension to our framework. A morphing of graphical objects is not possible.

The animation framework introduced in this paper is suited to the needs of syntax directed editing i.e. graphical objects that are modified may change the layout and position of other objects. This is detected automatically and DEViL's animation framework can interpolate this change on its own without the need of the animation designer. Hence, the animation specification is very easy and the animation automatically derived from the simulation specification is often everything the animation designer needs.

6. Conclusion

This paper proposed a new approach to animated visual languages in the context of a generator framework. The language DSIM can be used to specify the simulation of a VL. It uses concepts that were adapted by well-known simulation languages. Furthermore it introduces a simulation model that fits the needs of visual language simulation. The simulation model can be adapted to the needs of the simulation, e.g. it can add classes or attributes, it can define priority queues or random numbers to fulfill queue based simulation and it can define context sensitive functions which are needed in transportation simulations. The animation is automatically generated by the simulation through typed modification actions. This leads to a formal mapping between simulation and animation. The animation functions themselves are computed through linear interpolation and lead to a smooth and advanced representation. This is new, because many generator frameworks for syntax directed editors that support simulation lack the smooth animation. The introduced *"animated visual patterns"* help to customize the animation in a declarative way. They can be combined and parameterized to reach even complex and attractive animations. The animated visual patterns as well as the simulation language make excessive reuse of existing concepts in DEViL so that the language designer does not have to learn a completely new concept. This is reached by the reuse of classes, attributes, inheritance and path expressions in DSIM and the specification concept of a pattern library on the animation side.

References

[1] R. Bardohl. GenGed: A generic graphical editor for visual languages based on algebraic graph grammars. In *1998 IEEE Symp. on Visual Lang.*, pages 48–55, Sept. 1998.

[2] M. Brown and R. Sedgewick. A system for algorithm animation. In *Proceedings of ACM SIGGRAPH*, pages 177–186, 1984.

[3] E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by abstract state machines: The light control case study. 6(7):597–620, 2000.

[4] G. catalin Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.

[5] B. Cramer, D. Klassen, and U. Kastens. Entwicklung und evaluierung einer domänenspezifischen sprache für sps-schrittketten. In D. Fahland, D. A. Sadilek, M. Scheidgen, and S. Weißleder, editors, *DSML*, volume 324 of *CEUR Workshop Proceedings*, pages 59–73. CEUR-WS.org, 2008.

[6] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Universität Berlin, 2006.

[7] P. A. Fishwick. *Simulation Model Design and Execution*. Prentice Hall, 1995.

[8] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, Feb. 1992.

[9] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[10] U. Kastens. An attribute grammar system in a compiler construction environment. In *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 380–400. Springer Verlag, 1991.

[11] Microsoft. Die AsmL Sprache, 2009. http://research.microsoft.com/en-us/projects/asml/.

[12] B. A. Myers, R. C. Miller, R. Mcdaniel, and A. Ferrency. Easily adding animations to interfaces using constraints. In *In Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '96*, pages 119–128, 1996.

[13] C. D. Pegden, R. E. Shannon, and R. P. Sadowski. *Introduction to Simulation Using SIMAN*. McGraw Hill, 1995.

[14] M. J. V. Pereira and P. R. Henriques. Visualization/animation of programs based on abstract representations and formal mappings. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 373, Washington, DC, USA, 2001. IEEE Computer Society.

[15] E. Pietriga. A toolkit for addressing hci issues in visual language environments. pages 145–152, Sept. 2005.

[16] C. Schmidt, B. Cramer, and U. Kastens. Usability evaluation of a system for implementation of visual languages. In *Symposium on Visual Languages and Human-Centric Computing*, pages 231–238, Coeur d'Alène, Idaho, USA, Sept. 2007. IEEE Computer Society Press.

[17] C. Schmidt, P. Pfahler, U. Kastens, C. Fischer, and O. K. Gmbh. 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.

[18] T. J. Schriber. *An introduction to simulation using GPSS/H*. John Wiley & Sons, Inc., New York, NY, USA, 1991.

[19] H. D. Schwetman. Introduction to process-oriented simulation and csim. In *Proceedings of the 1990 Winter Simulation Conference*, 1990.

[20] XML Path Language (XPath) Version 1.0, W3C Recommendation, Nov. 1999. `http://www.w3c.org/TR/xpath`.