Electronic Communications of the EASST
Volume 32 (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

From the Behavior Model of an Animated Visual Language to its Editing Environment Based on Graph Transformation

Torsten Strobl, Mark Minas, Andreas Pleuß, Arnd Vitzthum

13 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

From the Behavior Model of an Animated Visual Language to its Editing Environment Based on Graph Transformation

Torsten Strobl[1], Mark Minas[1], Andreas Pleuß[2*], Arnd Vitzthum[3]

[1] [Torsten.Strobl,Mark.Minas]@unibw.de, Univ. der Bundeswehr München, Germany
[2] Andreas.Pleuss@lero.ie, Lero, Univ. of Limerick, Ireland
[3] Vitzthum@informatik.tu-freiberg.de, Technische Univ. Bergakademie Freiberg, Germany

Abstract: Animated visual models are a reasonable means for illustrating system behavior. However, implementing animated visual languages and their editing environments is difficult. Therefore, guidelines, specification methods, and tool support are necessary. A flexible approach for specifying model states and behavior is to use graphs and graph transformations. Thereby, a graph can also represent dynamic aspects of a model, like animations, and graph transformations are triggered over time to control the behavior, like starting, modifying, and stopping animations or adding and removing elements. These concepts had already been added to *DiaMeta*, a framework for generating editing environments, but they provide only low-level support for specifying and implementing animated visual languages; specifying complex dynamic languages was still a challenging task. This paper proposes the *Animation Modeling Language* (*AML*), which allows to model behavior and animations on a higher level of abstraction. AML models are then translated into low-level specifications based on graph transformations. The approach is demonstrated using a traffic simulation.

Keywords: animated visual language, behavior modeling

1 Introduction

Visual modeling languages (VLs) are widespread in engineering and computer science. Several frameworks and tools have been realized that make implementing VLs, i.e., providing tool support for such models, easier. *DiaGen*/*DiaMeta* [7], GenGED [1] or AToM^3 [6] are only a few of them. Many VLs have an execution semantics, i.e., models can be executed and executions are visualized by animations, e.g., *Statecharts* [3], *Pictorial Janus* [4], or *ToonTalk* [9] and many further examples, especially in simulation. However, there is still a lack of tool support, so dynamic VLs usually have to be implemented manually. Recently, a new approach for specifying interactive dynamic VLs based on graph transformation (GT) has been proposed and realized within the *DiaMeta* tool [13]. Previous approaches based on GT (e.g., [1]) use graphs for representing static model states whereas the effects of GTs can be animated. In contrast, graphs in [13] do not necessarily represent the static aspect of a model, but rather its dynamic aspects. GTs, when triggered at specific points of time, modify such graphs and implement the dynamic behavior of

* This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre, http://www.lero.ie/

the system. As a consequence, GTs can start, change, or stop animations, for example. GTs can easily describe interactions within the model or between user and model as well. This approach allows specifying dynamic VLs including rather complicated animations and interactions, e.g., several concurrent animations may simultaneously take place in a model. However, GTs without further abstraction mechanisms do not provide sufficient support for the easy specification of complex animations. This paper addresses this problem and proposes the *Animation Modeling Language* (*AML*), which is based on previous work with the Multimedia Modeling Language (MML) [11] and the Scene Structure and Integration Modeling Language (SSIML) [15]. Its purpose is the modeling of behavior and animations on a higher level of abstraction. It allows to decompose a dynamic system into its basic constituents and to describe behavior by hierarchical automata. *AML* models can then be refined and transformed into a specification for *DiaMeta* following the approach presented in [13].

The rest of the paper is structured as follows: The next section introduces a traffic simulation as the running example. Section 3 then briefly outlines the specification approach for animated interactive VLs presented in [13]. *AML* and the translation of *AML* models into a specification for *DiaMeta* based on GT are described in Sections 4 and 5. Section 6 reports on related work, and Section 7 concludes the paper.

2 Running Example: *Traffic*

Traffic simulations can be considered as complex dynamic systems. The modeling of the behavior for each traffic participant in such simulations is non-trivial because participants have to take care of many different situations. For this running example, simplified but still complex aspects of a traffic simulation have been chosen and realized in a system called *Traffic*. Fig. 1 shows two screenshots of a *Traffic* editor whose code has been generated based on a *DiaMeta* specification. A video of the resulting editor can be found online[1].

The road network in this system contains the following components: Roads, Intersections and EntryExit points. Thereby, a Road always connects two of the other components. It is not necessarily straight, i.e., it can also be curved. An Intersection always has a predefined size and one connection point for each cardinal direction. Finally, there are EntryExit points which are special elements where cars can enter or exit the simulation.

Each Intersection has one 4-state TrafficLight (states Go, Caution, Stop, and Ready) for each direction. The duration of states Go and Stop is configurable by a property interval available for each Intersection, and as a special feature for interactivity, the user is also allowed to click on an Intersection in order to trigger an immediate switch. Each EntryExit "produces" cars randomly. A parameter randomNext specifies the maximal amount of time between two cars. Cars that arrive at an EntryExit are "consumed" by the EntryExit and disappear.

Cars can accelerate and brake, but they have a fixed maximum speed. Cars must stop at a traffic light if it is in states Stop or Caution as long as there is enough time for stopping. At a predefined distance in front of each Intersection, each Car decides (randomly) whether to turn left or right or to move straight on and indicates its aim by turn signals. Cars have to obey apparent rules before and while turning, e.g., left-turning drivers have to wait when oncoming traffic blocks the road.

[1] http://www.unibw.de/inf2/DiaGen/animated

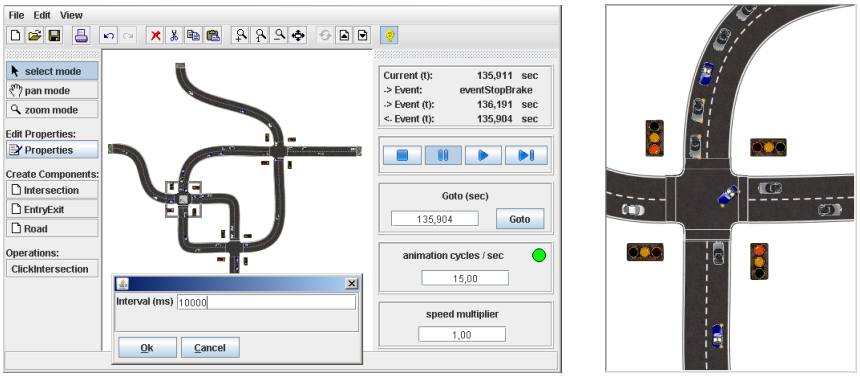

Figure 1: Screenshots: (a) editor during animated simulation, (b) zoomed

In addition, cars also have to watch cars in front of them and must start braking if the car in front is stopping and if the safety distance is violated otherwise. If the front car is starting again, a minor delay time for restarting the car behind is applied in order to imitate real world flow of traffic. Finally, Cars also have to watch traffic jams, i.e., they must stop in front of an Intersection if the destination street is jammed.

The remainder of the paper describes how the *Traffic* editing environment can be specified, focusing on the dynamic aspects.

3 Animation by Graph Transformation with *DiaMeta*

The GT-based approach for specifying animated interactive VLs described in [13] uses hypergraphs for internally representing animated visual models. Each model component (e.g., a Car, a Road, or an Intersection in the *Traffic* example) is represented by a *component hyperedge* that visits the nodes representing the component's attachment points. These hyperedges carry attributes representing properties of their component, e.g., its position. Model hypergraphs also contain *relation edges* (binary hyperedges), that stand for relationships between components, and further *link hyperedges* which have multiple purposes as shown later. The visual model is just a view of the hypergraph and depends on the hypergraph's attributes, but also on the continuously proceeding time. This means that a hypergraph without changing its structure or its attributes may still represent a visual model that is currently animated, i.e., the hypergraph represents both the current structure of a visual model and its current animation state. Changes of the structure and animation state are the result of events, which may be triggered externally, e.g., by the user, or internally, e.g., when a running car must start breaking because of a red light. With regard to such events, our approach is closely related to *DEVS* (cf. [14]).

External as well as internal events are GT programs (GTPs) consisting of GT rules together with a control program that modify the hypergraph of the animated visual model together with its attributes. However, GTPs representing external events are handled differently from GTPs representing internal events: A GTP that represents an external event is performed immediately as soon as the external event is triggered, e.g., when the user pushes a button. An internal event,

in contrast, occurs after a certain amount of time depending on the current structure of the visual model and its animation state. Therefore, the specification of an internal event consists of a GTP and a *time calculation rule*. This special rule is used for determining the point of time when such an internal event occurs: Whenever the hypergraph of a visual model is changed due to an event, the runtime system has to check which internal events may happen next. This is done by examining the GTPs of all internal events and checking whether they are enabled. However, the enabled transformations are not yet actually performed. Instead, the time calculation rules are used to compute the points of time when the events will occur. The GTP of the earliest internal event is actually performed at the computed point of time if no external event has been triggered meanwhile, changing the model's hypergraph and possibly removing other scheduled internal events. This procedure of computing the next internal event is repeated after each modification of the model's hypergraph.

This specification approach has been realized within the *DiaMeta* tool and has been used for several animated VLs as described in [13]. However, this specification approach can hardly be applied to more complex animated VLs without analyzing required structures and events first. The large amount of required events (i.e. GTPs), which appear confusing, is a inhibition threshold for realizing the VL specification. The following section introduces the more abstract modeling language *AML*, which addresses this problem.

4 Animation Modeling Language

Because of the complexity of an animated VL like *Traffic*, we propose to use an appropriate modeling language to specify animated VLs and their behavior for *DiaMeta*. This section introduces the Animation Modeling Language *AML*, which is based on previous work with two other modeling languages MML [11] and SSIML [15]. The long-term goal of *AML* is to define general concepts for modeling interactive animations which can be applied in other modeling languages, e.g., for multimedia or augmented reality. In the context of this paper, *AML* is used for a high-level specification of animated VLs from which the GT-based implementation can be derived. A simplified metamodel of *AML* is shown in Fig. 2 which presents *AML* as extension of UML elements.

The main structural elements in *AML* are visual elements. We adopt the concept of *media components* [11] to model them. A media component encapsulates some media content together with basic functionality, e.g., methods that render or play the media content, such as audio, video, 3D graphics. However, we will focus on 2D graphics in the following. Each media component provides some standard properties and operations depending on its media type, e.g., size and position for graphics. In addition, custom properties, operations, and associations can be specified in the same way as for UML components. For instance, Fig. 3 shows the graphics component for Intersection and its custom property interval, but without custom operations. Standard properties and operations need not be specified explicitly in the model.

A media component usually consists of several parts, e.g., a video is composed of multiple images or a graphic consists of different shapes. These inner parts can be important for the application's behavior: Complex animations often consist of multiple parts, such as a moving car whose turn signals should blink when it aims to turn. This requires that the graphics of the car's

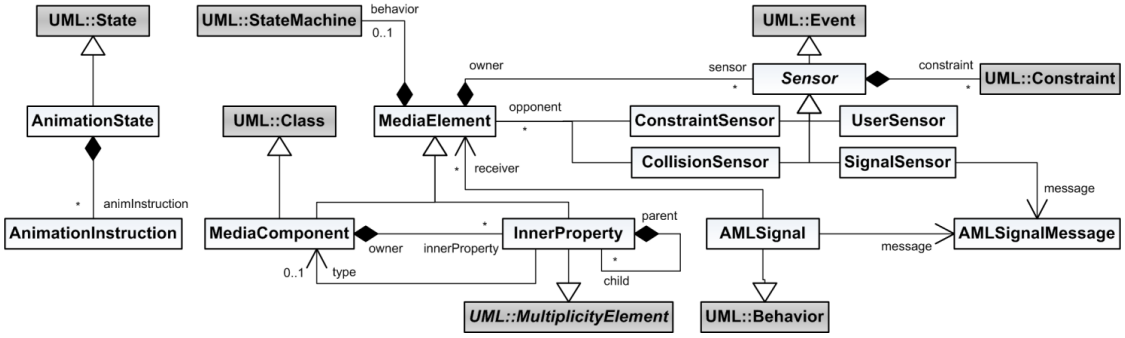

Figure 2: *AML* metamodel (simplified)

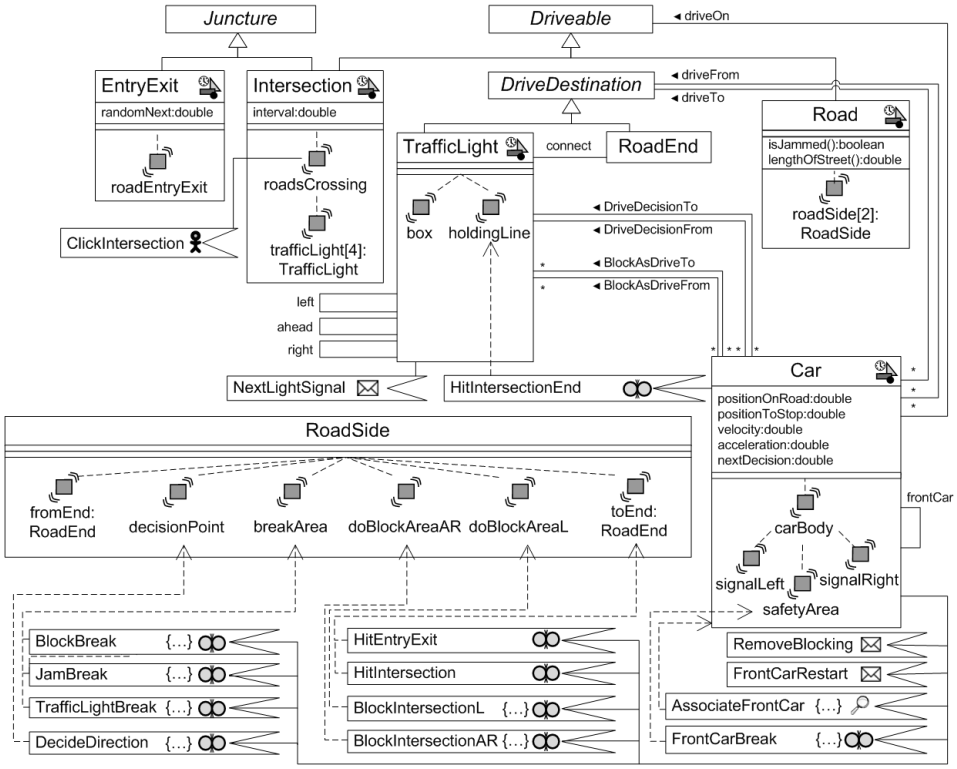

Figure 3: *AML* model: media components, associations, inner properties and sensors

turn signals are not part of the car's graphic itself, but attached graphics which can be accessed at runtime. Moreover, distinction between different parts is also important for event handling: For instance, the user might trigger different functionality when clicking on the intersection itself or when clicking on one of its traffic lights. Therefore, the inner structure of media components needs to be defined in *AML*.

A media component's inner structure is defined in terms of *inner properties* which are organized in a hierarchical manner. Manipulations of parent properties also affect their children; e.g., moving a car also means moving all its inner properties (e.g., the turn signals). An inner property has a name, an optional type, and an optional multiplicity. A type needs only to be specified to indicate that this inner part is an instance of another media component. If no type is specified, the media part is just an instance of an anonymous media component. A multiplicity can be specified to indicate multiple instances. For instance, the Intersection graphics in Fig. 3 consists of a roadsCrossing which contains four trafficLights. The latter ones are instances of another media component TrafficLight while for roadsCrossing no further type is specified.

Dynamic behavior of animations is modeled using specific kinds of events. In conventional interactive applications, events are mainly triggered by user actions, such as pressing a button. However, applications with complex animations require additional kinds of events resulting from the dynamic behavior of media components. For instance, behavior should be triggered if a moving object reaches a specific position or touches another object on the screen (e.g., a car reaches a traffic light) or if a certain point of time is reached.

In *AML*, this is modeled by different kinds of *sensors*. Four kinds are presented in this paper. Fig. 3 shows several sensors which are denoted similarly to an accept event action in UML. Fundamentally, a sensor is owned by a media component or by a media component's inner property. A *user sensor* listens for user events, such as clicks on a specific component. For instance, ClickIntersection listens for a user click on roadsCrossing which is a part of an Intersection. A *signal sensor* such as NextLightSignal is similar, but is used for passing events internally, so media components can pass messages to each other. A *collision sensor* has a relationship to one or more other graphic components (called *opponents*) and triggers an event when its owner collides with its opponent(s), i.e., when they overlap on the screen. The collision sensor DecideDirection owned by the graphic component Car triggers an event as soon as the car reaches a specific area on its street. As a consequence, the driver decides upon a direction and starts indicating if necessary.

A sensor can also be associated with OCL constraints to specify that the sensor is active only under certain conditions. Most sensors in Fig. 3 contain constraints, but details are not illustrated there. Several additional keywords such as owner, opponent (sensor) or parent (inner property) are available in constraint expressions allowing to refer to involved components or to navigate through their structures. A special kind of sensors, the *constraint sensor*, is always modeled with OCL expression. The purpose of this sensor type is to observe connected components for satisfying the OCL constraint. An example is sensor AssociateFrontCar which checks whether a car is driving behind another one on the same RoadSide and, in this case, associates them in the model by the association frontCar.

The behavior and animations of a media component or an inner property is modeled by a special kind of state machine (see Fig. 4). As different aspects of one component can have their own behavior, multiple state machines (or regions) can be executed in parallel. The state machines basically support the same concepts as state machines in UML, i.e., states, pseudo states, parallel states, state transitions, guards, and activities associated with states or transitions. Expressions in the state machines can refer to all properties and operations of its owner.

The most important triggers for transitions are the sensors (see above). The sensor's name can be denoted at the transition which means that the transition is performed when the sensor triggers an event. In addition to events effected by sensors, it is also possible to use *elapsed time events*

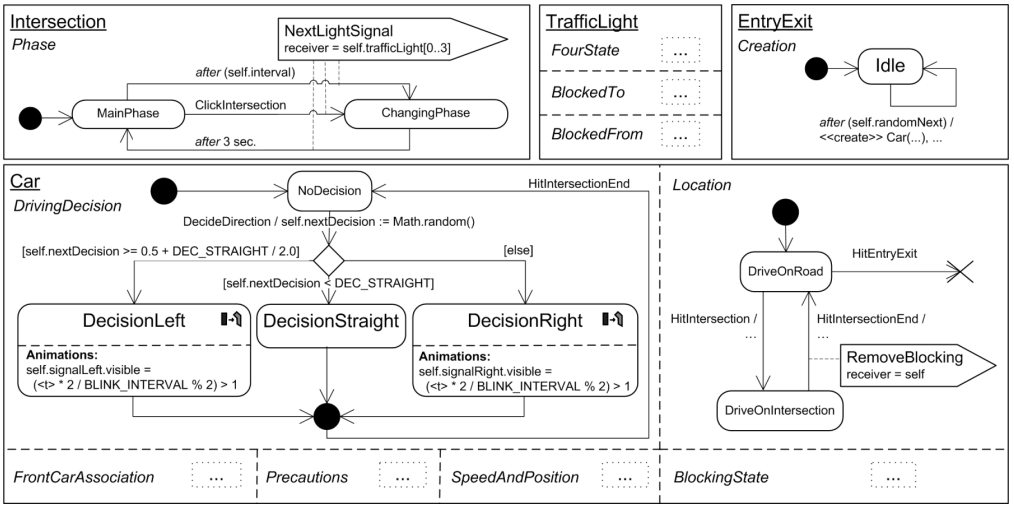

Figure 4: *AML* model: state machines and animation behavior (excerpt)

or *change events* as trigger for transitions. The former is indicated by keyword after and occurs automatically after a period of time in the state, the latter is indicated by when and occurs as soon as annotated constraints are satisfied.

Finally, transitions can also send signals to other components. In *AML*, this is denoted explicitly by a special kind of send signal action where one or more receivers of the signal can be specified explicitly (using the keyword receiver). Such signals are designed for corresponding signal sensors of the receiver. After receiving a signal, these sensors trigger other events, if possible. An example in Fig. 4 is NextLightSignal which is sent by an Intersection to each of its TrafficLights at the same time one of the attached state transitions is performed.

The most important *AML*-specific concepts are *animation states* which are a special kind of state. They define the change of properties over time while the owner of the state machine is in this state (i.e., the animation of the graphic component or a part of it). Animation states have a small symbol in the top right corner and *animation instructions* at the bottom. Within these instructions, properties of media components, e.g., position or angle, can be bound to expressions. Fig. 4 shows two examples of animation states. For instance, state DecisionLeft describes the blinking of the car's left turn signal by switching the visibility attribute of graphic component SignalLeft on and off depending on the elapsed time (denoted by $<t>$) and a constant for the interval.

5 Translating *AML* Models to *DiaMeta* Specifications

AML models describe all dynamic aspects of an animated VL. This section describes how a *DiaMeta* specification (called "specification" in the following) can be derived from an *AML* model. So far, there is no automatic translation process from model to specification. Instead, a model is manually translated into a specification using the following five steps (see Fig. 8).

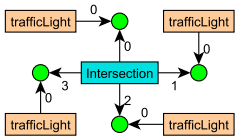

Figure 5: Intersection and TrafficLight

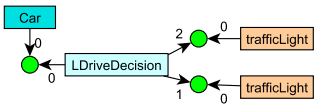

Figure 6: Car associated with TrafficLights

(1) Specifications of static components and dynamic components of the VL represented by *AML* media components must be derived. (2) The resulting specification must be extended by constructs which allow the representation of all states within the *AML* model. (3) GT rules must be derived from the model in order to map each state transition. (4) Resulting GT rules must be specified as *DiaMeta* events. (5) Animation visualizations during animation states must be considered.

Each of the following subsections describes one step for *Traffic*. Please note, that only a rough picture of these steps can be presented due to the space limitations of this paper. The translation starts with the specification of static VL components (1.1) and dynamic VL components (1.2).

(1.1) As a first step, the *AML* media component model (see Fig. 3) must be investigated for static language elements. While independent media components are declared as regular VL components, media elements that are used as inner properties must be specified as *sub-components*, i.e., they are represented by so-called *sub-component (hyper-)edges* being connected to the component hyperedge of its parent component through appropriate nodes. Fig. 5 shows an Intersection as a component hyperedge and its trafficLights as sub-component edges.

(1.2) The *AML* model usually contains further media components (e.g., Car) and associations that are used during animation only. Such elements are maintained by state transitions of the behavioral model. For example, *AML* associations belonging to dynamic aspects require the specification of so-called *animation (hyper-)edges* which can link components in order to represent associations. Fig. 6 shows animation edge LDriveDecision which corresponds to both *AML* associations DriveDecisionFrom and DriveDecisionTo. The shown graph represents a car whose driver has decided to drive from the first traffic light attached to LDriveDecision to the second one.

(2) The next step is to translate all state machine states (see Fig. 4). Thereby, the state of the dynamic system must be expressible by the graph representing the model (cf. Sec. 3), i.e., specifications which allow the characterization of each component's state are required. In some cases, no additional specification is necessary because a state can already be determined implicitly (e.g., by attached animation edges). The state of a Car's state machine DrivingDecision, for instance, is already characterized by edge *LDriveDecision* (see Fig. 6). Depending on its connections to trafficLight edges, it is determined whether the driver wants to drive straight, left, right, or has not decided yet if there is no such edge.

A specification option, which is always available for embedding state information, is to create an explicit attribute for a (sub-)component edge. This attribute has to store the active state.

(3) Next, state transitions modeled in *AML* are analyzed in order to derive GT rules that, when executed, realize the behavior of the *AML* state machines. As described above, states are encoded in the hypergraph of an animated model. A GT rule simulating a transition from state A to state B,

hence, must be enabled if the state machine is in state A and all guard conditions of the transition are satisfied (which includes guards of triggers such as sensors). As a result, the rule has to modify the hypergraph model such that it contains the encoding of state B. In the simplest case, switching from state A to B means changing the state attribute of a component. In addition, the established GT rule also has to perform actions described by the *AML* state transition.

(4) Each state transition shown in the last section is triggered by events. As explained in Sec. 3, two kinds of events can be distinguished and specified for *DiaMeta*, so the following subsections address the specification of external events (4.1) and internal events (4.2).

(4.1) In *AML*, events are observed by sensors, i.e., user sensors such as ClickIntersection in case of external events. In terms of *DiaMeta*, the specification of an external event means the specification of a GTP which is executed directly if the user selects a component and pushes a defined key or button. This event-related GTP, and the following is also true for internal events, must choose the applicable state transition depending on the recipient's current state and execute the corresponding GT rule specified in step (3).

In a sense, but not correct in terms of definition, receiving signals (e.g., NextLightSignal) can also be considered as external events from the recipient's point of view. Therefore, signal sensors can be translated into external event specifications in the same way. However, the specified GTPs may not be accessible from outside of the system, e.g. for the user.

(4.2) The *DiaMeta* specification of internal events, which are also based on GTPs, initially follows the guidelines described in the previous step, except that internal events can never be accessed by an external system or the user. In addition, a time calculation rule is required for internal events and must be specified. In order to derive meaningful time calculation rules, different *AML* elements indicating internal events are described in the rest of this step:

The simplest internal events found in the *AML* models are *elapsed time events* indicated by keyword *after*. In this case, the GTP is already complete, and the required time calculation rule can be deduced directly by adding the argument of the elapsed time event, e.g., "3 sec", to the time the state has been entered. This implies that a *state entry time* must be tracked at corresponding state transitions, e.g., by maintaining an appropriate component attribute. Another kind of internal events are *change events* indicated by keyword *when*. They can also result in a simple time calculation rule, i.e., the rule that always returns the time when the internal event is calculated. This means that the state transition is performed immediately if the graph pattern and additional conditions match, which is possible if arguments of the change event are part of the GT rule's graph pattern and conditions. However, if a condition depends on a value which changes during an animation state, the time calculation rule, which has to calculate the first point of time when the condition is true, gets more complicated.

Finally, constraint and collision sensors trigger internal events as well. Both are similar to change events, but they provide a more convenient notion for observing the relationships of multiple components. The GTP of the derived event specification is usually based on the involved (sub-)components' hyperedges, observed constraints and relationships determined by the sensor. Thereby, a collision sensor is a special kind of constraint sensor, which implies colliding components as additional constraint. This constraint is usually related to movement animations, which are visualized during animation states (i.e., between state changes). In such cases, a time calculation rule, which calculates the point of time when the first collision happens, is required. An exemplary collision sensor is DecideDirection, which must be translated into three internal event

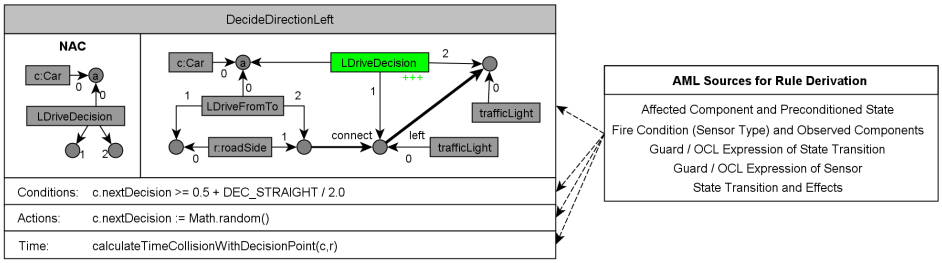

Figure 7: Event specification for DecideDirectionLeft and according state transition

specifications. Depending on the Car's attribute nextDecision only one of them may be applicable. Fig. 7 shows the rule DecideDirectionLeft[2] which is applied if this attribute indicates the decision to drive left. The figure also outlines the *AML* elements used to derive the rule.

(5) The remaining part is the interpretation of the *animation states* within the *AML* model, e.g., a Car which is moving along a Road or a Car with an activated turn signal signalLeft (see Fig. 4). The animation instructions stated in animation states indicate which and how the attributes of the component must be changed while an animation state is active (cf. Sec. 4). These instructions must be transferred into the *DiaMeta* specification. In terms of *AML*, property changes are performed while in a state without any transition necessary. For *DiaMeta* this means that graphical primitives are not drawn using static attributes, but drawn using temporarily calculated values derived from such attributes and modified by referring to the current animation time and the associated animation instruction. Finally, when leaving or entering an animation state, it is often necessary to update static attributes which have been the basis for animation visualization before. Within the *DiaMeta* specification, this must be done by the GT rule which realizes the corresponding state transition. For instance, a Car is moved during an animation state called Drive. The Car's position (static attributes) must be updated when leaving this state. Otherwise, information about the new position would not be available after the movement animation.

6 Related Work

AML integrates concepts from two existing modeling languages: the *Scene Structure and Integration Modeling Language* (SSIML) for 3D development and the *Multimedia Modeling Language* (MML) for interactive multimedia applications. SSIML/Behaviour [15] is a modeling language based on UML2 state machines which also introduced animation states for modeling animation details but focused on 3D applications.

MML [11] is a platform-independent language for model-driven development of multimedia applications. It provides concepts such as complex media components with inner structure, user interface elements, and sensors, which have been reused here. However, it does not support

[2] It shows the GT rule within one segment: parts which are added by the rule are drawn in green with "+++". Attribute conditions and modifications are illustrated by expressions within the two extra boxes *Conditions* and *Actions*. The time calculation rule is shown in box *Time*.

modeling animations yet and will be extended with concepts from *AML* in the future.

In the area of interactive multimedia, there are only few other modeling approaches so far. OMMMA [12] allows modeling multimedia applications including media objects. Dynamic behavior is specified by statecharts while static animations are modeled by extended UML sequence diagrams. However, the models are more abstract than in MML and do not directly support code generation. [5] presents a modeling approach for computer games illustrated by a tank game. It uses statecharts on multiple abstraction layers for different aspects which are specially suited for game design: sensors, memorizers, strategical deciders, tactical deciders, and actuators.

AML supports separated behavior specifications of different components interacting in a common environment. This possibility makes *AML* also attractive for the specification of domain-specific visual languages for agent-based modeling and simulation. In this field of research other tools are already available, e.g., in [10] a toolkit for specifying the behavior of agents and their visual appearance within a modeling environment is demonstrated.

There are also other concepts for the animation of VLs based on GT. An example is the approach described in [1] which basically allows specifying animations for discrete event simulations. However, animations are visualized for state transitions (GTs) which restricts the VLs especially in terms of interactivity and parallelism of independent animations. The resulting animations are self-running movies, and amalgamated GT rules are already required for the specification of less complex examples such as animated Petri nets.

In [8] a set of visual languages is introduced in order to describe the behavior of diagrams of metamodeled languages. The concepts are also based on events and state machines, but most languages are less close to UML. In addition, the concepts include the modeling of user interfaces, whereas aspects of smooth graphical animations are not covered. Concerning the flexibility, the approach is less adequate for a language with many independently animated actors (agents) such as in *Traffic* as the behavior is represented by a single state machine.

Another approach for the application of models in order to implement graph-based simulations is shown in [14]. The authors describe how DEVS models can be used as a semantic domain for programmed GTs which allows simulation-based design. However, the concepts describe GTs consuming time and because of possible needs for parallel executions or interruptions, additional control structures are required.

7 Conclusions

In this paper we introduced the *Animation Modeling Language* (*AML*) to model complex animations at an appropriate abstraction level. It has been used here to systematically derive *DiaMeta* specifications to generate interactive editors for animated VLs.

Different levels of details can be chosen, so *AML* can illustrate rough ideas as a starting point for *DiaMeta* specifications, but also models which are very close to the resulting specifications. On the other hand, fully featured *AML* diagrams even document *DiaMeta* specifications which are usually less explanatory. Using *AML* and *DiaMeta* is a promising approach for simplifying the specification effort for complex VLs.

We now aim at an automated transformation of customized *AML* diagrams into *DiaMeta* specifications, so further research objectives include platform-specific extensions of *AML* allowing

for the automated generation of graph-based VL specifications. In this context, a reasonable set of predefined animation instructions must also be found and offered by our framework in order to avoid self-programmed routines realizing animations or collision detection.

Bibliography

[1] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Tech. Univ. Berlin, Books on Demand, Norderstedt, 2006.

[2] S. Gyapay, R. Heckel, D. Varró. Graph Transformation with Time: Causality and Logical Clocks. In *ICGT '02*, LNCS 2505, Springer, 2002, pp. 120–134.

[3] D. Harel, H. Kugler. The Rhapsody Semantics of Statecharts (or, on the executable core of the UML). In *Integrations of Software Specification Techniques for Applications in Engineering*, LNCS 3147, Springer, 2004, pp. 325–354.

[4] K. Kahn, V. Saraswat. Complete Visualizations of Concurrent Programs and their Executions. In *1990 IEEE Workshop on Visual Languages*. 1990, pp. 7–15.

[5] J. Kienzle, A. Denault, H. Vangheluwe. Model-based Design of Computer-Controlled Game Character Behavior. In *Models 2007*, LNCS 4735, Springer, 2007, pp. 650–665.

[6] J. de Lara, H. Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modeling. In *FASE '02*, LNCS 2306, Springer, 2002, pp. 174–188.

[7] M. Minas. Generating Meta-Model-Based Freehand Editors. In *GraBaTs'06*. ECEASST 1. 2006.

[8] T. Mészáros, G. Mezei, H. Charaf. Engineering the Dynamic Behavior of Metamodeled Languages. In *Simulation* 85(11):793–810, 2009.

[9] L. Morgado, K. Kahn. Towards a specification of the ToonTalk language. In *J. of Visual Languages and Computing* 19:574–597, 2008.

[10] M.J. North, E. Tatara, N.T. Collier, J. Ozik. Visual Agent-based Model Development with Repast Simphony. In *Agent 2007 Conf.*. 2007.

[11] A. Pleuß. MML: A Language for Modeling Interactive Multimedia Applications. In *7th IEEE Int. Symp. on Multimedia (ISM'05)*. IEEE, 2005, pp. 465–473.

[12] S. Sauer and G. Engels. UML-based Behavior Specification of Interactive Multimedia Applications. In *IEEE Symp. on Human-Centric Computing Languages and Environments (HCC 2001)*. IEEE, 2001, pp. 248–255.

[13] T. Strobl, M. Minas. Specifying and Generating Editing Environments for Interactive Animated Visual Models. In *GT-VMT'10*, ECEASST 29. 2010.

[14] E. Syriani, H. Vangheluwe. DEVS as a Semantic Domain for Programmed Graph Transformation. In *Discrete-Event Modeling and Simulation: Theory and Applications*. CRC Press, 2009.

[15] A. Vitzthum. SSIML/Behaviour: Designing Behaviour and Animation of Graphical Objects in Virtual Reality and Multimedia Applications. In *7th IEEE Int. Symp. on Multimedia (ISM'05)*. IEEE, 2005, pp. 159–167.

A Appendix

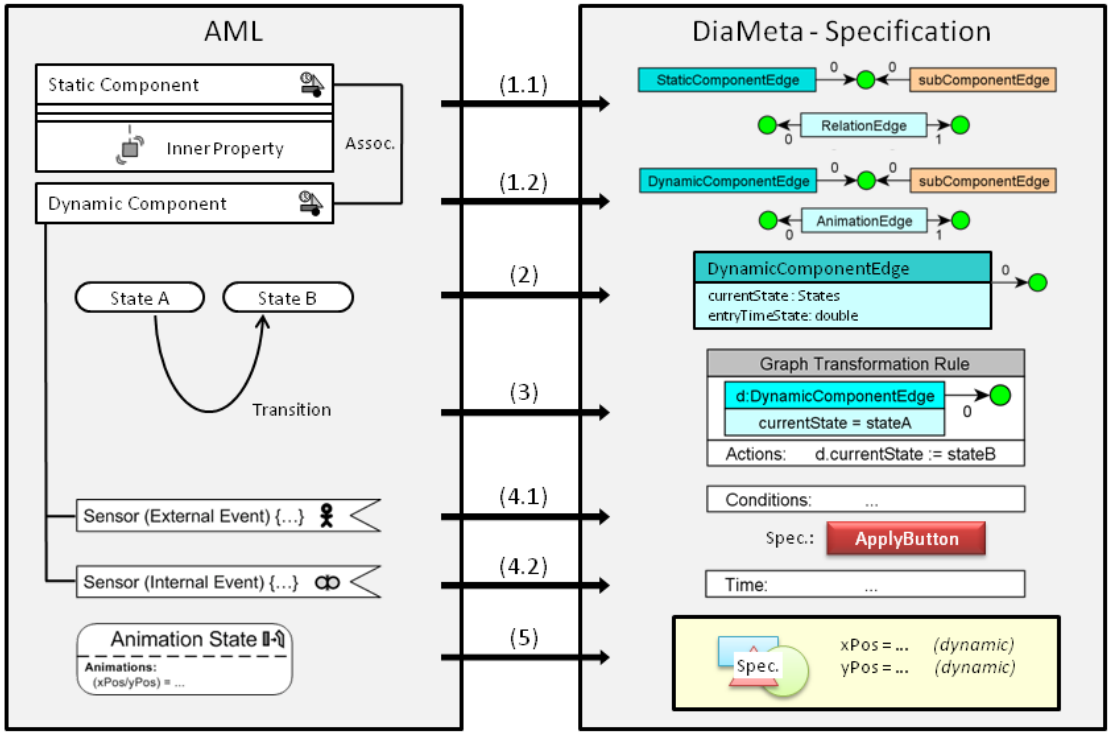

Figure 8: Overview of the translation steps