Generating Graph Transformation Rules from AML/GT State Machine Diagrams for Building Animated Model Editors

Torsten Strobl and Mark Minas

Universität der Bundeswehr München, Germany
{Torsten.Strobl,Mark.Minas}@unibw.de

Abstract. Editing environments which feature animated illustrations of model changes facilitate and simplify the comprehension of dynamic systems. Graphs are well suited for representing static models and systems, and graph transformations are the obvious choice for implementing model changes and dynamic aspects. In previous work, we have devised the Animation Modeling Language (AML) as a modeling approach on a higher level. However, AML-based specification could not yet be translated into an implementation automatically. This paper presents a language extension called AML/GT and outlines how AML/GT models can be translated into graph transformation rules automatically and also provides some implementation details.

1 Introduction

Visual models are considered to be important tools of software development. Particularly, the area of domain-specific languages (DSLs) is an interesting topic for research and industry. Therefore, some tools like *GenGEd* [3], *AToM*^{3} [6] or *DiaGen*/*DiaMeta* [8] support the creation of editors for DSLs with little effort. For this purpose, many of these tools generate editors out of mostly text-based editor specifications and use graphs for representing models internally, together with graph transformations (GTs) for changing them.

However, visual models are not restricted to static structures. They can also contain execution semantics, for instance the popular example of Petri nets. Editors for such models usually support animated simulations.

This paper continues our work extending the *DiaMeta* toolkit in order to facilitate the implementation of editors for complex animated models and languages with preferably minimal effort. Our first step was to allow the specification of event-based model changes through graph transformation rules (GTRs) [15]. It is based on the idea of a static graph representing an animated model that changes while time passes by. Graph transformations happen instantaneously, and they can be used for starting, stopping or modifying animations, whereas other approaches (e.g., *GenGEd* [3]) represent animations by graph transformations that do not happen instantaneously, but last as long as the animation takes.

Case studies showed that such GTR-based specifications are yet too unstructured for a convenient specification of animated systems. Therefore, in a second

A. Schürr, D. Varró, and G. Varró (Eds.): AGTIVE 2011, LNCS 7233, pp. 65–80, 2012.

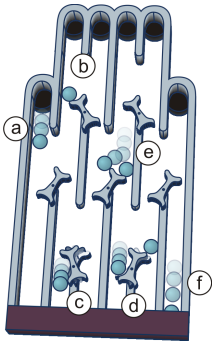

Fig. 1. Avalanche board

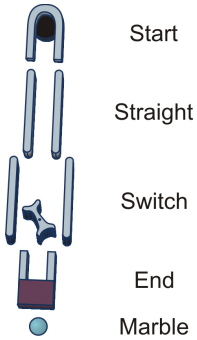

Fig. 2. Avalanche pieces

step, the modeling language AML (Animation Modeling Language) has been introduced for supporting structured design and specification of animated systems [16]. The language offers, among other features, the behavioral specification of individual components and a convenient way for describing animations for particular states. In that way, AML is a helpful tool for creating editor specifications manually, but it was not yet possible to automatically generate GTR-based specifications from AML-based specifications. This paper closes this gap and introduces AML/GT as an extension of AML as well as a tool that automatically transforms an AML/GT-based specification into a GTR-based specification for *DiaMeta* which can then be used to generate the implementation of the animated system. As a running example, an editor called Avalanche is created.

The rest of the paper is structured as follows: Section 2 outlines the running example Avalanche. Section 3 describes how animated editors can be realized with *DiaMeta* in short. AML is revisited in Section 4 and its extension AML/GT is introduced in Section 5. The translation of AML/GT state machines into GTRs is elaborated in Section 6. Section 7 gives some implementation details. Finally, Section 8 shows related work, and Section 9 concludes the paper.

2 Avalanche

This section sketches Avalanche (see Fig. 1) which has been implemented as a model editor and serves as example throughout this paper. Although Avalanche is based on a board game, gaming aspects are ignored and it is treated as an interactive dynamic system. A detailed view on Avalanche and its implementation using *DiaMeta* is presented in [15]. In addition, an animated example can be found online.[1]

In Avalanche marbles are falling down the lanes of an inclined board. The board itself can be built by four types of block pieces (see Fig. 2). The top

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

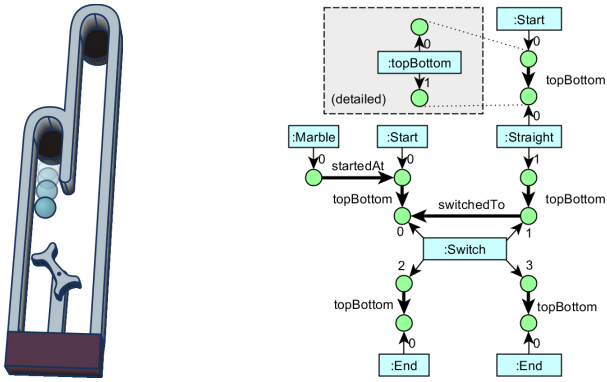

Fig. 3. Avalanche board and corresponding hypergraph

of each lane is limited by a *Start* piece. A *Marble* can be placed there, and it starts falling as shown in Fig. 1 (a) then. While falling, the *Marble* can be stopped by the upper side of a *Switch* (b). On the other hand, if a *Marble* hits the bottom side (c), a *Switch* is tilted to the neighboring lane. In this case, a previously stopped *Marble* can be released (d). It is also possible that a *Marble* hits another *Marble*, which is currently blocked by a *Switch*. Then, the falling *Marble* bounces off and changes its lane (e). Finally, a *Marble* can reach an *End* piece where it is removed from the board (f).

3 Specification of Animated Model Editors

Avalanche shall be implemented as animated model editor which allows the creation of individual boards (e.g., the board in Fig. 3). The user can put marbles in *Start* pieces. Afterwards, they start falling, and the animated model shows the game mechanics. Important aspects are the interaction between marbles and switches and that new marbles can arrive at any time, for example. The code for the Avalanche editor shall be generated from a specification for the *DiaMeta* system. Therefore, this section introduces some basics of *DiaMeta* and the applied generation process.

Internally, models of *DiaMeta* based editors are represented by **typed, attributed hypergraphs**. *DiaMeta* supports three types of (hyper)edges in such hypergraphs. **Component hyperedges** represent model components. These edges visit nodes that represent the component's **attachment areas**. Such areas may be spatially related, e.g., overlap, which is represented by so-called **relation edges** between the corresponding nodes. Relation edges are created or removed automatically whenever attachment areas of components start or end being spatially related [8]. Finally, **link hyperedges** can be used for connecting nodes of component hyperedges. Such edges can be created and removed explicitly.

In Fig. 3 an Avalanche board is shown together with its internal hypergraph. Because of such underlying graphs, it is possible to specify model changes via GTs. Our approach of implementing animated editors is also based on GTs, so a brief description follows. A detailed elaboration, however, can be found in [15], which also includes the underlying formalism.

The internal graph does not necessarily **represent a static model**. Instead, a constant graph may describe how visible components are **animated**, e.g., by determining where, when and how a component started moving. Then, the visualization of the graph can involve the current animation time for illustrating an animated scene. The hypergraph in Fig. 3 actually shows a marble with appended *startedAt* edge, which stands for a currently falling marble during the visualization of the model.

Besides such (graphical) animations, there are also **time-dependent model changes** caused by GTs. These GTs are the result of internal and external events. **External events** are sent to the system from an external source, e.g., from a user sending a command to the editor, whereas the occurrence of **internal events** is based on the system state. Such events are specified (and implemented) by GTRs, or rather graph transformation programs in *DiaMeta*. They are equipped with application conditions to trigger them when external events happen, or with a **time calculation rule (TCR)** that computes the time when the internal event occurs. This calculation usually depends on the attribute values of the related components.

However, the behavior of the Avalanche editor is not specified directly via GTRs and event details as described above in this paper. Instead, the modeling language AML/GT (see Sections 4 and 5) is used and GTRs will be generated.

4 Animation Modeling Language (AML)

The presented specification approach comes along with the Animation Modeling Language (AML). This section provides an overview focused on aspects relevant for this paper. Some more details, origin, and goals of AML are presented in [16].

AML models describe both the structure and relations of graphical components and their dynamic behavior including animations. The language is based on UML 2.3 [11]. AML refines UML class diagrams for the static structure and UML state machine diagrams for the dynamic part (see Fig. 4).

The following extensions are available for the static structure:

- **Media Components** are the main structural element of AML. They extend regular UML classes by aspects of visual components. They always include basic attributes like *xPos*, *yPos*, *angle* and other attributes which determine spatial aspects or drawing state.
- **Inner Properties** are similar to UML properties, but their specified type must be a media component. In addition, they can be arranged hierarchically. In this way, inner properties can model the compositions of graphical sub elements within their containing media components.

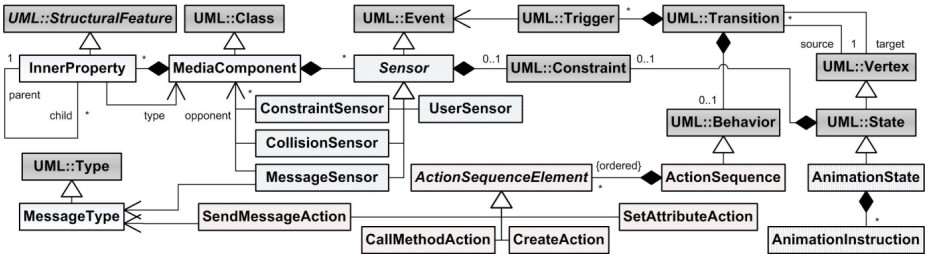

Fig. 4. AML metamodel (excerpt)

– **Sensors** are owned by media components and observe states, animations and interactions of its owner.

The three most important kinds of sensors are:

– **User Sensors** are *triggered* if the user interacts with the sensor owner in a specific way, e.g., if the user clicks on a component.
– **Constraint Sensors** are *triggered* if its guard expression (by default an OCL expression [10]) evaluates to *true*, which may also happen during an animated situation. **Collision Sensors** are a special subtype of constraint sensors. They can observe when the owner collides with another component, called **opponent**.
– **Message Sensors** are *enabled* if the sensor's owner receives a message of a declared type from another component, called **opponent**, too.

Fig. 5 shows an AML/GT model, i.e., an AML model with GT extensions described in the next section, but it illustrates plain AML elements, too. The inner property *lever* ❷,[2] e.g., represents the graphical lever within the media component *Switch* ❶ which represents the whole board piece. It is also shown how the default value of the *lever*'s attribute *angle* is redefined there. The user sensor *PutMarble* ❸ is *triggered* if the user clicks on a *Start* component, which is the sensor owner ❹ indicated by the solid line. The collision sensor *MarbleEnd* ❺ is *triggered* if its owner *Marble* collides with the bottom of an *End* piece, which is connected via a dashed **opponent arrow** ❻.

Behavior and animations of media components are described by state machine diagrams and their regions ❽. Each state machine models the states and transitions of components of one media component type.

For the description of graphical animations, AML supports special states called **animation states** ❾ that contain so-called **animation instructions** ❿. Each animation instruction computes the value of an attribute of the state machine's media component while time passes by. E.g., animation instruction ❿ defines that the angle of a *Switch* lever changes from $-30°$ to $30°$ within three seconds as soon as the *Switch* enters state *LeftRight*.

[2] Numbers in black circles refer to the ones in Fig. 5.

Transitions can be labeled like in regular UML by *trigger[guard]/effect*. The **trigger** describes an event that may cause the transition. Intuitive keywords like *at* or *after* (11) indicate certain points in time or time delays. AML also supports sensor triggers specified by the name of the sensor (12) that must be owned by the state machine's component. The transition is *enabled* when the associated sensor is *triggered*. Finally, transitions without trigger specification (13) are *triggered* as soon as the state's internal activities (e.g., active animation instructions) terminate (*completion event*).

A **guard** may restrict transition triggers; a transition is only *enabled* if the guard condition is satisfied, too, like in UML.

In AML, effects as well as *entry* and *exit* behaviors of states are restricted to so-called **action sequences** consisting of an arbitrary series of the following actions:

- **Set actions** (14) can change an attribute of the corresponding component or one of its inner properties.
- **Create actions** (15) allow the creation of media components.
- **Call actions** (16) are used for calling operations.
- **Send actions** can send messages to other media components.

The **execution of transitions** is performed sequentially. Since a system with discrete time model may have multiple *enabled* transitions at the same time and the UML specification does not define full semantics of conflicting transitions, *enabled* AML transitions are executed with the following priority, starting with the highest one: sensor events, time events, without trigger/with guard, without trigger/without guard. In addition, each sensor can have an explicitly specified numeric priority value. Such values must be used to order a set of simultaneously occurring sensor events. If there are still conflicts, a transition must be chosen randomly.

5 Animation Modeling Language for GTs (AML/GT)

AML enables developers to model Avalanche and to translate it to a GTR-based *DiaMeta* specification manually as described in [16]. For an automated translation, however, AML models lack important information. Therefore, AML has been extended to AML/GT which addresses this issue. Technically, AML/GT is realized as a (UML) profile for extending AML models. This section describes some of its elements in the context of the Avalanche model shown in Fig. 5.

Some media components represent visual components that the user can place on the screen using the generated Avalanche editor. They are marked by a small square next to the media component icon, e.g., *Switch* (1), and are then represented by *component hyperedges* (see Section 3). In addition, inner properties of media components can be marked by small circles (17). Such properties are represented by nodes of this component hyperedge. Inner properties which model graphical aspects of the component are marked by a small eye (2).

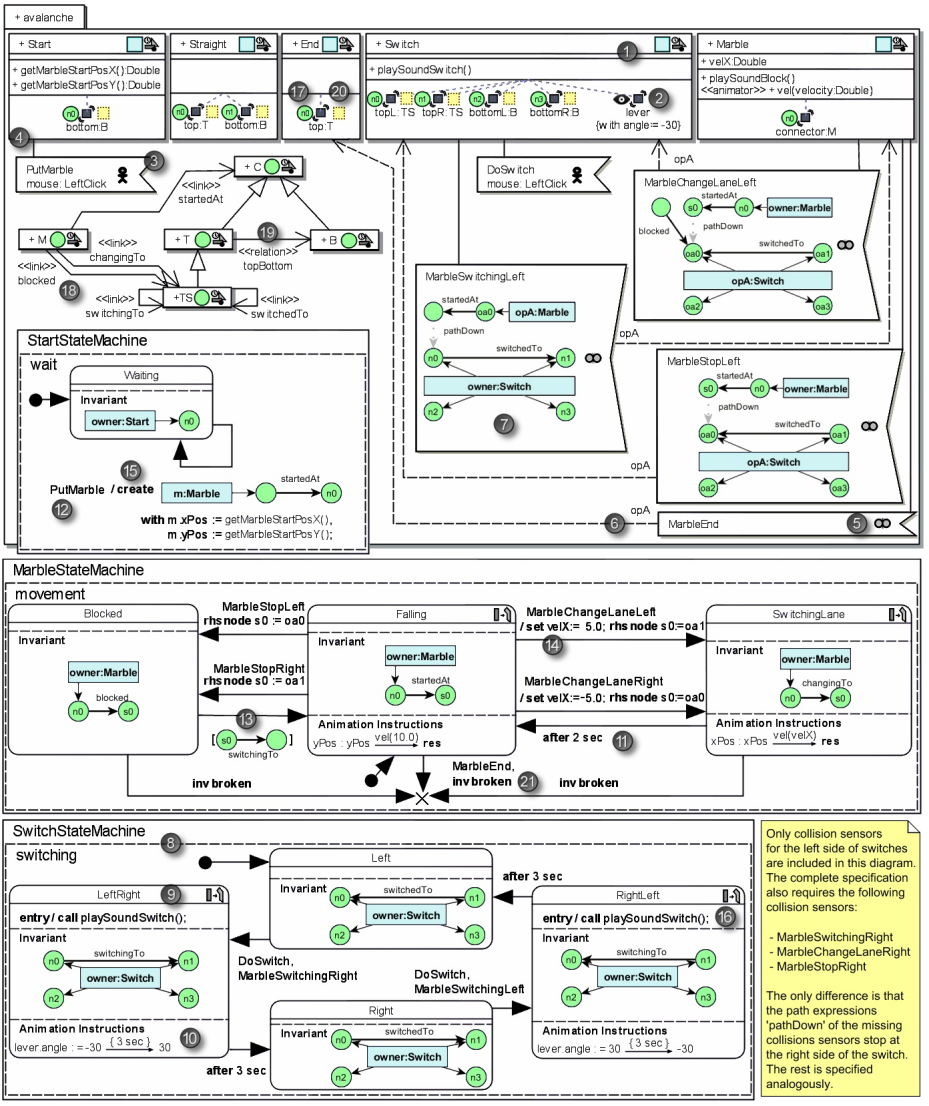

Fig. 5. Avalanche specification (AML/GT model)

A *Switch* hyperedge, for example, visits the nodes *topL*, *topR*, *bottomL* and *bottomR* which represent the four corners of the corresponding component. The types of these nodes are *TS* ("top/switch") or *B* ("bottom"), resp., as specified in Fig 5 ❶. Fig. 5 also shows how attributes of the inner property's media component type are redefined by new default values ❷.

Moreover, associations between media components can be interpreted as edge types which can be used to connect the nodes of component hyperedges. Associations corresponding to *link hyperedges* are marked with stereotype **link** ⓲.

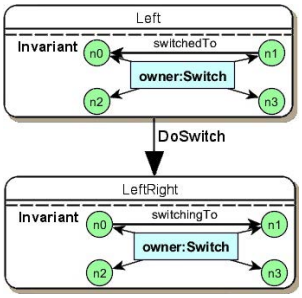

Fig. 6. Exemplary transition for *Switch*

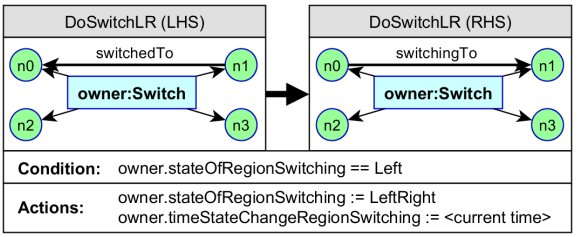

Fig. 7. GTR for *DoSwitch* transition

Relation edges indicating spatial relationships between attachment areas of visual components correspond to binary associations with stereotype **relation** (19). The specification of such an attachment area is marked by a small square (20).

6 Translating AML/GT State Machines into GTRs

After the basic introduction of AML/GT, this section shows how AML/GT models are translated into GTRs. These GTRs become part of the specification of the animated editor which is then used by *DiaMeta* to generate the editor as described in Section 7.

Each AML/GT state machine describes the behavior of a media component, and each of its states corresponds to a subgraph of the animated diagram's hypergraph. It contains the component hyperedge of the media component and some additional link hyperedges. This subgraph represents the situation when the modeled media component is in this state, i.e., such a subgraph, called **invariant pattern** in the following, represents an invariant of the state and is visually denoted inside the state box. For example, the state *Left* in Fig. 6 contains the link edge *switchedTo* next to the component hyperedge itself. By connecting node *n1* with node *n0*, the edge indicates that the switch's *lever* is currently blocking the *Switch*'s left lane. In the same figure, there is also a transition to state *LeftRight*. Fig. 7 shows the GTR realizing this state transition.

In order to create the required set of GTRs, the algorithm has to iterate over all transitions with or without triggers (a completion event trigger is assumed in the latter case). During this process, the triggers have **priorities** according to the rules at the end of Section 4. Each GTR of the resulting set can be specified with this priority then. GTs with higher priority can be processed first, if there is more than one GT scheduled at the same point in time.

The following steps show how single GTRs are created. They are based on the example in Fig. 8 and its example trigger *MarbleChangeLaneLeft* (1). The illustrated transition is responsible for changing a *Marble*'s state from *Falling* to *SwitchingLane*, i.e., the *Marble* has to move to another lane because it has hit the left side of a *Switch* which is currently blocked by another *Marble*. The graph

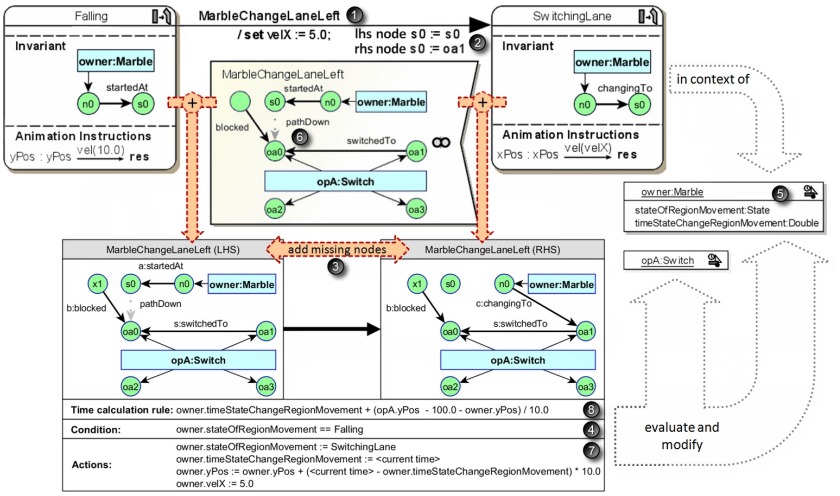

Fig. 8. Generating a GTR from a state machine transition

pattern shown within the collision sensor *MarbleChangeLaneLeft* represents this condition.

The **LHS** of the rule is created first. It consists of the invariant pattern of the transition's source state, extended by subgraphs of the transition's guard (e.g., Fig. 5 ⓭) as well as graph patterns describing the behavior of sensors being used as transition triggers (e.g., Fig. 5 ❼). In the example shown in Fig. 8, there is no transition guard, but the graph pattern located within collision sensor *MarbleChangeLaneLeft* has to be added. In general, the LHS is constructed from several graph patterns. Negative application conditions are possible, too, but they are not required here.

As a basic principle, the creation of the LHS from all these graph patterns is performed by building the union of them and then gluing them in a suitable way. Gluing is rather straight forward because each of the graph patterns refers to the same "owner" of the state machine which is represented by its component hyperedge and its visited nodes. Therefore, the LHS may contain the component hyperedge of its owner only once, i.e., all instances of this hyperedge and its visited nodes must be glued. Further nodes can be glued as well. In the example, node *s0* of the source state graph pattern must be glued with *s0* of the pattern within the sensor. This is specified by the **correspondence statement** *lhs node s0* := *s0* ❷. However, because equally labeled nodes are glued automatically, such a statement is omitted in Fig. 5.

The **RHS** of the rule is built in a similar way. It is constructed from the invariant of the transition's target state and, again, all graph patterns of the transition as for the LHS. The latter have to be added, too, because applying the rule shall just change the state of the owner's state machine, i.e., change its invariant pattern; the rest of the diagram's hypergraph must remain unchanged.

The owner's component hyperedge again determines which edges and nodes must be glued. The gluing of an additional node is specified by the correspondence

statement *rhs node s0 := oa1* ❷. The node labeled *s0* of state *SwitchingLane* does obviously not correspond to the node with the same label of state *Falling*; it should rather correspond to the node *oa1* representing the switch's attachment area where the marble is switching to as soon as the specified transition is executed.

The rule must not add or delete any nodes since each node represents an attachment area of a component, and animations do not add or remove media components (except if created or deleted explicitly; see below). Therefore, nodes without correspondence in either LHS or RHS must be added to the other side ❸.

After creating the main parts of the GTR, additional elements must be added, i.e., further application conditions, a TCR necessary for scheduling events, attribute changes, and maybe other types of processing.

Further application conditions are usually expressions which check attribute values of graph elements. For this, the AML/GT model can contain OCL expressions (conditions), or other Boolean expressions (Java) in case of *DiaMeta*. Such expressions must be adapted to a syntax which is compatible to the GT system and added to the GTR accordingly.[3]

As described above, the source state's invariant pattern is part of the corresponding GTR's LHS. However, just relying on the invariant pattern of the source state is generally not sufficient as the *inv broken* trigger shows (see below). Therefore, a **state attribute** is added to the component hyperedge ❺ which is checked before the rule may be applied ❹. There must be one such attribute for each UML region because of concurrent and hierarchical states.

Another type of application conditions are **path expressions**. In Fig. 8, a path expression called *pathDown* is required ❻. It verifies that the *Marble* is currently falling down the lane which leads through the blocked left side of the *Switch*. In *DiaMeta* this expression is specified as an arbitrary sequence of *topBottom(0,1)*, *Switch(0,2)*, *Switch(1,3)*, or *Straight(0,1)* edges. The numbers within the parenthesis specify the hyperedge tentacles the path must follow: the first number specifies the ingoing tentacle and the second one specifies the outgoing tentacle when following the path through the hypergraph.

AML/GT allows specifying **actions** at different places. They may be specified with each transition; they are executed as soon as the transition fires (e.g., Fig. 5 ⓮) or as an action associated with an entry into a state (e.g., Fig. 5 ❾). Actions are easily translated into GTRs in *DiaMeta* since it allows arbitrary Java code when executing a rule. *Call actions* result in the calls of media component operations (generated Java code; e.g., Fig. 5 ❾), and *set actions* involve the change of attributes of the component hyperedge during the GT (e.g., Fig. 5 ⓮). Finally, *create actions* can construct components (e.g., Fig. 5 ⓯): AML/GT allows declaring component hyperedges (here *Marble*) including its initial attributes, its nodes, and link edges. These edges and nodes must be added to the RHS of the GTR, and attributes must be set accordingly.

[3] Single graph transformation rules are not sufficient in the following; graph transformation programs are actually needed. However, for simplicity, we still use the term "graph transformation rule" (GTR) instead of "graph transformation program".

Further actions must be added to each GTR (Fig. 8 ❼). First, the *state attribute* and the **state entry time** must be updated (such an attribute must be available for each region again). And second, the attributes which are considered *animated* during an active animation state must be updated because attributes do not change their values during animations in our approach (cf. Section 3). Instead, the changing value is calculated using a formula considering the animation time. Therefore, an update which applies the last calculated value is required.

A generated GTRs must be executed at a specific point in time which is modeled by a transition's **trigger**. A transition triggered by a user sensor is translated into a GTR that can be induced by the user in the editor. By default, the GTR is bound to a GUI button starting the GTR if it is applicable. Other sensor triggers require a **TCR** ❽ for scheduling the GT (see Section 3).

Transitions with time event triggers are translated into GTRs representing *internal events* with a TCR reflecting the absolute or relative time when the event may be triggered. Relative time always refers to the *state entry time*.

Transitions without explicit trigger must be executed as soon as the pattern of the LHS can be matched and animations of the source state have been finished (an additional condition). They are translated into GTRs representing *internal events* as well. Its TCR must return the point in time when animations have been finished. If there is no animation, the GTR must be applied without any time delay, i.e., the TCR always returns the current time.

Constraint and collision sensors result in internal event specifications, too. They usually check *animated* attributes (see above). However, generating a corresponding TCR is not straight forward. For a collision sensor, e.g., a TCR is required which calculates the collision time based on the components' trajectories. Currently, such a TCR has to be provided manually. Using a physics engine may solve this problem in the future.

Finally, two special cases for generating GTRs from the Avalanche model are described. **Termination states** are used as the only means to delete component hyperedges and, therefore, media components. Such states are visualized as a small X. If a transition ends in a termination state, it is translated into a GTR that removes the corresponding component hyperedge and its nodes, so the resulting RHS corresponds to the LHS without the component's edge and its nodes.

Inv broken is an AML/GT keyword that may be used as a transition trigger. It has lowest priority and is *fired* if the transition's source state must be left because its invariant pattern is "broken". This may happen if some link edges being part of the invariant pattern are deleted when some other component and its link edges are deleted due to a user action.

For instance, a *Switch* may be removed by the user even if a *Marble* is connected to one of its nodes via *startedAt* (state *Falling* in Fig. 5 ㉑). Because the specification in Fig. 5 requires a marble to be connected to the component where it started falling, the *Marble* would be in an inconsistent state then. The *inv broken* trigger is used to represent the fact that the pattern invariant of the current state is suddenly no longer true although the state has not yet been left.

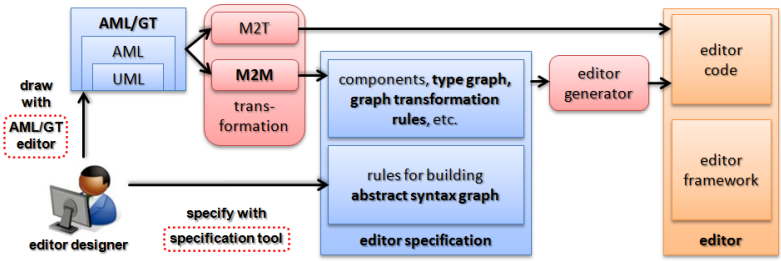

Fig. 9. Overview of the editor generation process

This triggers the state's *inv broken* transition and activates its termination state which removes the *Marble*, too.

The translation of such an *inv broken* transition into a GTR is straight forward: The GTR must check the component hyperedge's *state attribute* (see above) and whether the invariant pattern is violated. The latter is simply represented as a negative context, i.e., the GTR may be applied if the component is in the corresponding state, but its invariant pattern can (no longer) be matched.

In order to support all features of AML/GT state machines, further special cases must be considered when generating GTRs as well. Namely, **composite states**, **concurrency** and **message passing** must be processed specifically. These topics are not discussed here, but it is clear that they can be translated to GTRs, too.

7 Implementation

The translation process outlined in the previous section has been completely implemented, also covering those topics that have been omitted here, e.g., composite states, orthogonal regions or message passing. It is possible now to specify animated editors (e.g., for Avalanche) using a visual specification tool (the AML/GT editor) and to generate the editor from this specification. This section sketches the process, which is outlined in Fig. 9, and some design decisions.

For the creation of AML/GT models, the editor designer uses the AML/GT editor (see Fig. 10), which has been generated using the *DiaMeta* toolkit, too. AML/GT models are automatically translated into the specified animated editor. Some Java code of the editor is directly generated from the AML/GT model ("M2T" in Fig. 9). This is necessary for support code which has to be provided manually when using *DiaMeta* without AML/GT. The "M2T" transformation has been realized with *Acceleo*, a Model-to-Text translation language which is oriented towards the **MOFM2T** specification of the OMG [9].

The translation process from an AML/GT model into an editor specification for *DiaMeta* ("editor specification" in Fig. 9) as outlined in the previous section is performed by the "M2M" component in Fig. 9. It uses *Eclipse QVTo*, which implements the OMG specification of **MOF QVT Operational** [12],

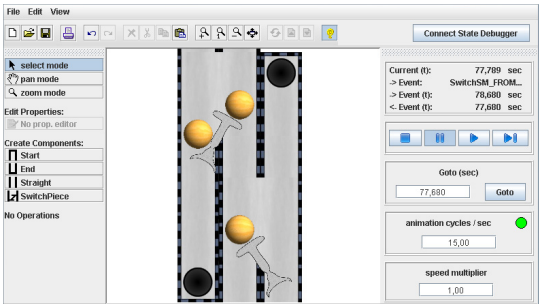

Fig. 10. Screenshot of the AVALANCHE editor

a Model-to-Model technology. The target model of this transformation is the native, XML-oriented *DiaMeta* specification which also includes the GTRs.

There are several reasons for choosing these technologies instead of using the *DiaMeta* framework for creating the specification from AML/GT models. First of all, the GT system of *DiaMeta* has been created for structured editing in editors and therefore lacks convenient features for complex model transformations. In addition, both source (AML/GT) and target model (*DiaMeta* specification) do not consist of graphs only, but also contain embedded text or must produce Java code which are less suited for being modeled as graphs. Finally, standardized languages have been favored in order get a more future-proof, general and stable transformation. The mentioned languages have already been supported by a couple of tools (Eclipse plugins), too.

Besides the AML/GT model, the editor designer has to provide the abstract syntax[4] of the animated visual language, i.e., its meta-model. *DiaMeta* simply uses EMF for metamodelling. Any EMF-based tool for specifying Ecore models [14] can be used here. Furthermore, generating TCRs from collision sensors still needs manual code (see Section 6), so future solutions require a generic *a priori* collision detection algorithm.

Finally, the "editor generator" creates the Java code of the animated editor using the code generator of *DiaMeta*. The screenshot in Fig. 10 shows the generated AVALANCHE editor with created AVALANCHE board and a falling marble. This editor has been generated from the AML/GT model shown in Fig. 5.

8 Related Work

Although it has not been the primary intention of AML/GT, it can be considered as a visual language for programmed graph rewriting and model transformation. A good starting point for reading into this topic is provided in [4], which compares *AGG*, *Fujaba*, and *PROGRES* as graph transformation languages and also mentions *GReAT*. Another overview is presented in [18] which focuses

[4] All descriptions in this paper and AML/GT concern the concrete syntax only.

on model transformation aspects including the tools *AToM*^{3}, *VIATRA2*, and *VMTS*. However, AML/GT cannot compete with these rather general-purpose languages and tools because the GTRs generated are restricted to the very specific field of animation specification. For example, a regular transition can only add or remove link hyperedges. In general, it is specialized for describing the behavior of components.

Furthermore, there are many other systems and languages which are also intended for describing behavior, especially in the context of statecharts, UML, and MDE, e.g., Executable UML (xUML). Many of such languages try to extend or constrain UML in order to get a language with the possibility to describe systems more precisely than UML. However, many of them have a specific application domain. AML and AML/GT, which also extend and constrain standard UML, focus on visual appearance and graphical animations next to the expressiveness of UML statecharts and hypergraphs. Therefore, they are especially suited for specifying animated visual languages and generating editors. To the best of our knowledge, they are the only languages featuring this combination, so the following related work has more or less other domains.

Fujaba, for example, supports so-called story diagrams [2] which combine UML collaboration diagrams with activity diagrams. GTRs can be drawn within activities which allow the creation of complex transformation flows. Originally, the main purpose was to generate Java code from such models, but areas of application evolved and many extensions are available in the meantime. For example, *Fujaba Real-Time* [1] allows the modeling of embedded systems based on statecharts with real-time capabilities. The modeling of visual aspects of components, however, is not provided. On the other side, AML/GT is not suited for embedded systems and does not provide the modeling of real-time constraints.

Another modular and hierarchical model-based approach is presented in [17]. The semantic domain of the models presented there is the Discrete Event System Specification (*DEVS*). It is used for describing control structures for programmed graph rewriting. Although the formalism has a solid foundation, it requires the user to have a specialized expertise while UML-based approaches are well-known to the majority of all users. Moreover, it also lacks the possibility to define the structure of components or graphic related issues.

In [13] a visual language for model transformations and specifying model behavior has been introduced which allows the specification of in-place transformation rules. These rules can be compared to GTRs in a graph-based environment. There are some further extensions like rule periodicity, duration, exceptions, etc. State-based modeling on a more coherent level, which is also the intention of this paper, is not provided.

Finally, the animation approach used in this paper and the translation of state machines into GTRs is related to other approaches. The differences between our animation approach and those approaches in similar systems (e.g., [3]) have been discussed in previous work [15] already. Translating state machines into equivalent GTR systems is not a new idea either. E.g., there are several papers describing the semantics of (UML) state machines based on GTs, e.g., in [5].

The translation process described in Section 6 is rather tailored to its specific application within animated editors. Finally, the way graph patterns are used in state machines and sensors of AML/GT can be considered as alternative to OCL [10]. We also would like to point towards agent-based modeling [7] as another field of application for AML/GT.

9 Conclusions

We have pursued our approach of modeling animated editors with the UML-based language AML. The language extension AML/GT offers additional elements which are necessary to create specifications of dynamic systems in a hypergraph-based environment automatically.

Applying a modeling language like AML/GT promises that complex systems can be specified in a clear and accessible way. Using state machines for individual components particularly complies with an intuitive perspective on many systems. At the same time, the underlying GTs provide an established foundation and exact semantics, and they facilitate the comprehension of the execution process.

The algorithmic translation of AML/GT model into the specification format of *DiaMeta* has been completely implemented. The aim of creating a higher-level modeling language, which does not require further specifications for generating interactive, animated editors, has been accomplished. Future work will concentrate on how collisions between animated components can be detected without computing their trajectories in advance. Using a physics engine as used in many game settings appears to be promising.

References

1. Burmester, S., Giese, H.: The Fujaba Real-Time Statechart Plugin. In: Giese, H., Zündorf, A. (eds.) Proc. of the 1st International Fujaba Days 2003, pp. 1–8 (2003); Technical Report tr-ri-04-247, Universität Paderborn, Informatik
2. Diethelm, I., Geiger, L., Zündorf, A.: Systematic Story Driven Modeling, a Case Study. In: Giese, H., Krüger, I. (eds.) Proc. of the 3rd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM 2004), ICSE Workshop (2004)
3. Ermel, C.: Simulation and Animation of Visual Languages Based on Typed Algebraic Graph Transformation. Ph.D. thesis, Technical University Berlin (2006)
4. Fuss, C., Mosler, C., Ranger, U., Schultchen, E.: The Jury is Still Out: A Comparison of AGG, Fujaba, and PROGRES. In: Ehrig, K., Giese, H. (eds.) Proc. of the 6th International Workshop on Graph Transformation and Visual Modelling Techniques. ECEASST, vol. 6 (2007)
5. Kuske, S.: A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 241–256. Springer, Heidelberg (2001)
6. de Lara, J., Vangheluwe, H.: AToM^3: A Tool for Multi-formalism and Metamodelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)

7. Macal, C.M., North, M.J.: Tutorial on agent-based modeling and simulation. In: Kuhl, M.E., Steiger, N.M., Armstrong, F.B., Joines, J.A. (eds.) Proc. of the 37th Winter Simulation Conference, pp. 2–15. ACM (2005)
8. Minas, M.: Generating Meta-Model-Based Freehand Editors. In: Zündorf, A., Varró, D. (eds.) Proc. of the 3rd International Workshop on Graph-Based Tools. ECEASST, vol. 1 (2006)
9. Object Management Group (OMG): MOF Model To Text Transformation Language, v1.0 (January 2008), http://www.omg.org/spec/MOFM2T/1.0
10. Object Management Group (OMG): Object Constraint Language, v2.2 (February 2010), http://www.omg.org/spec/OCL/2.2
11. Object Management Group (OMG): Unified Modeling Language: Superstructure, v2.3 (May 2010), http://www.omg.org/spec/UML/2.3/Superstructure
12. Object Management Group (OMG): MOF Query/View/Transformation, v1.1 (January 2011), http://www.omg.org/spec/QVT/1.1
13. Rivera, J.E., Durán, F., Vallecillo, A.: A Graphical Approach for Modeling Time-Dependent Behavior of DSLs. In: DeLine, R., Minas, M., Erwig, M. (eds.) Proc. of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 51–55. IEEE Computer Society (2009)
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2009)
15. Strobl, T., Minas, M.: Specifying and Generating Editing Environments for Interactive Animated Visual Models. In: Küster, J., Tuosto, E. (eds.) Proc. of the 9th International Workshop on Graph Transformation and Visual Modeling Techniques. ECEASST, vol. 29 (2010)
16. Strobl, T., Minas, M., Pleuß, A., Vitzthum, A.: From the Behavior Model of an Animated Visual Language to its Editing Environment Based on Graph Transformation. In: de Lara, J., Varró, D. (eds.) Proc. of the 4th International Workshop on Graph-Based Tools. ECEASST, vol. 32 (2010)
17. Syriani, E., Vangheluwe, H.: Programmed Graph Rewriting with Time for Simulation-Based Design. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 91–106. Springer, Heidelberg (2008)
18. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., Varró-Gyapay, Sz.: Model Transformation by Graph Transformation: A Comparative Study. In: Proc. Workshop Model Transformation in Practice (Satellite Event of MoDELS 2005) (2005)