A Modular Approach to Animation of Simulation Models*

R. Bardohl [1], *C. Ermel* [1], *L. Ribeiro* [2]

[1] Technische Universität Berlin, Germany

[2] Universidade Federal do Rio Grande do Sul, Brazil

Abstract

The use of simulation within software development has as main aims to validate the system, as well as to recognize performance aspects. A simulation model includes the description of the behavior of the systems, but also intended animation strategies (to visualize the simulation) and statistics collecting/evaluating procedures. In this paper we present a modular approach for animating simulation models. Both the system's behavior and its animation are modeled by graph grammars, a formal, yet intuitive visual specification technique. The simulation of the system as well as its animation are supported by tools. We discuss the integration of the tools PLATUS, designed to construct the simulation models, and the tool GENGED, designed for the development of visual languages, in order to obtain an environment in which animation modules can be obtained (from GENGED) and used (in PLATUS). Moreover, due to the modular structure, an animation can be exchanged easily, and can be also reused by other simulation models.

Keywords: Simulation Model, Animation, Graph Grammars, Visual Languages.

1 Introduction

The area of discrete simulation aims at validating and recognizing behavior and performance aspects of a system. Although it is possible to perform tests on real systems, it poses pragmatical problems, especially during the design of safety critical system operations. Therefore, testing and validation of system properties during the software specification phase reduces costs and safety risks and helps to detect specification errors early. The use of discrete simulation allows to build controlled environments where different strategies may be tested while searching for the best solution.

*Research partially supported by the German Research Council (DFG) and by the projects GRAPHIT (DLR, Germany, and CNPq, Brazil) and PLATUS (CNPq, FAPERGS)

The first step to do a simulation of a system is to construct a model of the system to be simulated that represents the specific system elements under investigation as realistic as possible. A simulation model is composed of many parts. The main one describes the behavior of the system, others are concerned with collecting data for statistics and visualizing the simulation of the system. There are many interactive development environments for simulation, both commercial ones such as MicroSaint, Arena, Taylor II and ModSim III, as academic ones such as VMSS [KN96], SMOOCHES [BRMB96], VSE [Bal97] and SIMOO [Cop97]. In most of these environments, the behavior of the model, together with its animation and statistics components, is described by programs written in various programming languages. Describing the behavior of the components of a simulation system by programs has some serious drawbacks:

- Although the model is a specification of the system to be constructed, the description is too low level to be used as a specification for the further development of the system. Moreover, typically the programs describing the models include statistics and animation procedures and it is hard to extract the behavior model from this program (statistics and animation are only interesting for the simulation, but not for the development of the system);
- It makes it hard, if not impossible, to make proofs of behavioral properties of the system;
- The reuse of simulation components/animation procedures in other simulation models becomes more difficult;
- If the platform/implementation language changes, all existing models may be lost, or have to be adapted / re-implemented in a new language.

To overcome these problems we proposed in [CK98] that a simulation environment should have the following characteristics:

1. Use a formal specification language to describe the simulation models. This specification, besides being an abstract, implementation independent description of the system, can be used as a basis for verification and correct implementation.
2. Separate the aspects *Behavior*, *Statistics* and *Animation* throughout the modeling process. This separation allows flexible combinations of components (for example, the selection of different animation views) when simulating the system's behavior. Moreover, the analysis of system properties and the later software development steps then can be performed based on the *Behavior* component only, independently of animation or statistics details.

These ideas served as a basis for the definition of the PLATUS environment [RC00, CMR00]. This environment allows a component-wise construction of simulation models based on the formal specification technique *graph grammars* [Ehr79, Roz97]. Graph grammars are quite intuitive even for non-theoreticians. The basic idea is to model states as graphs and state changes as (graph) rules where a rule shows part of a model in a before/after style. This allows a clear and consistent description of events representing the system behavior.

Within PLATUS, the specification of the behavior of a component is clearly separated from the specification of the animation strategy chosen to visualize the simulation of this component, allowing to use a library of animation modules. One open problem was how to provide a user with a suitable way of constructing animation modules, and furthermore how to link these modules to the simulation model. This is the topic of this paper. To construct the animation modules we will use an approach called GENGED [Bar00]. Originally, the GENGED environment (short for *Generating Graphical Editors*) was built for the visual definition of visual languages. Here, we use it to construct animation modules. Animations will be described by graph grammars, that are the specification formalism for the description of visual languages in GENGED. We describe which kind of interfaces are necessary for the integration of the PLATUS components with the animation modules defined by GENGED (see Figure 1). The advantages of such a modular approach are twofold: on the practical side, the user deals only with one formal method in order to specify the behavior and the animation of his models; on the theoretical side, the uniform representation gives rise to simpler and more elegant models that are easier to analyze and to use for further development.

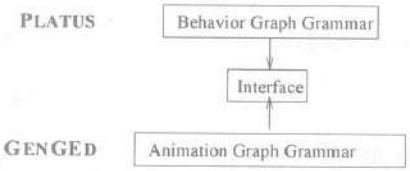

Figure 1: **Modular simulation and animation environment.**

This paper is organized as follows: Section 2 is an informal introduction to graph grammars as they are the basis formalism used in our approach; Sections 3 and 4 present the basic concepts of the simulation environment PLATUS and the GENGED approach respectively. In Section 5 , we show how to construct animation modules for simulation models. Finally, in Section 6 , we review the main results of this paper and discuss further developments.

2 Graph Grammars and Graph Transformations

Graphs play an important role in many areas of computer science. They are especially helpful in analysis and design of software applications like database systems or distributed systems. Prominent representatives for graphical notations are entity relation diagrams, message sequence charts, Petri nets, automata and all kinds of UML diagrams.

Now we will recall the concepts of typed graphs [Kor95, CMR96] and illustrate those by an example. A graph is given by two disjoint sets (graph objects), called nodes (vertices) and directed arcs (edges) from a source node to a target node. Every graph object is typed over a type graph. Figure 2 (a) represents a typed graph with three nodes and two arcs. The nodes are of type *Elevator*, *ButtonSet* and *press* and the arcs of type *solid line* and *dashed line*. The corresponding type graph is shown in Figure 2

(b). Here, the nodes and arcs are the types themselves, whereas the graph objects in Figure 2 (a) can be seen as instances of these types. Nodes and arcs may be additionally labeled by attributes that are used to store data together with the graph objects. In this paper we will only use attributes for nodes. Attributes will be denoted in the type graph by an arc carrying an attribute name connecting a node to its attribute type (a set). In the instance graphs this attribute arc will connect a node with the current value of that attribute. In Figure 2, the type graph (b) specifies that an *Elevator* node may contain an attribute named *state* of data type *State*= {stop, up, down}. In the instance graph (a) the value of this attribute is stop. The *press* node contains no attribute. We allow as attributes abstract data types, that is, we consider not only the sets of types, but also operations on these types. In particular, the use of abstract data types allows us to use variables and terms as attributes (by choosing a term algebra as attribute algebra). This is very useful for a high-level specification of behavior, as we will see later on.

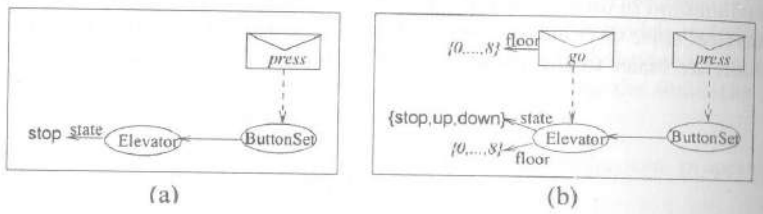

Figure 2: **A graph (a) typed over the type graph (b)**

Note that, using a graph as type system poses additional constraints on the possible instances: for example, in no instance graph a *press* node can be connected to another node via a solid line because such a situation is not present in the type graph of Figure 2 (b).

A relationship between graphs can be expressed by a graph morphism that maps the nodes and arcs of the first graph G to nodes and arcs of the second graph H, respectively. The graph objects in G are called *origins* and in H *images*. The mappings have to be *type compatible* (nodes and arcs are mapped to nodes and arcs of the same type) and *compatible with structure* (the source/target node of an arc is mapped to the source/target node of the arc's image). A graph morphism g between the graphs G and H is denoted by $g : G \to H$. Figure 3 shows a graph morphism. The attribute values also have to coincide. Note that the *press* node and its adjacent arc are not mapped. We call such a morphism *partial*. Morphisms mapping all objects in the origin are called *total*.

Graph transformation defines a rule-based manipulation of graphs[1]. Graph rules can be used to capture the dynamical aspects of systems. The resulting notion of graph grammars (consisting of a start graph and a set of graph rules) generalizes Chomsky grammars from strings to graphs. The start graph represents the initial state of the system, whereas the set of rules describes the possibles state changes that can occur in the system. A rule comprises two graphs: a left-hand side L and a right-hand side R, and a graph morphism $r : L \to R$ between the graph objects of L and R. Graph

[1]We here follow the Algebraic Single-Pushout approach to graph grammars [Low93, EHK+97]. For an overview of the main approaches see [Roz97].

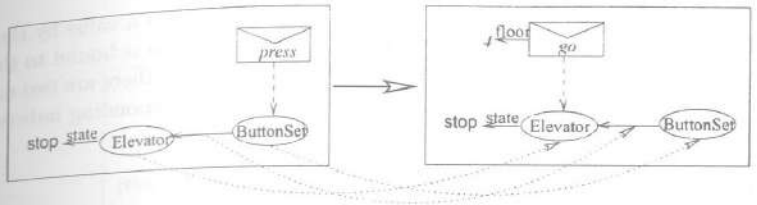

Figure 3: **A Graph morphism of attributed graphs**

objects in L which do not have an image via r in R are deleted; graph objects in R without original in L are created, and graph objects in L which are mapped to R by r are preserved by the rule.

The application of a rule to a graph G (derivation) requires a mapping from the rule's left-hand side L to this graph G. This mapping, called *match*, is a total graph morphism $m : L \to G$. A match marks the graph objects in the working graph that participate in the rule application, namely the graph objects in the image of m. The rule application itself consists of three steps. First, the graph objects marked in the rule for deletion are deleted. Thereafter, the new graph objects are appended to the graph. As a last step, all dangling arcs are deleted from the graph. The graph transformation results in a transformed graph H.

To achieve an abstract representation of behavior, rules often use variables and terms as attributes. Using attributed graphs, the attribute values or variables of the rule's left-hand side have to match as well. An attribute variable is bound to an attribute value in the mapped graph object by the match. In the transformed graph, the attribute values are evaluated depending on the rule's right-hand side and result in a constant value.

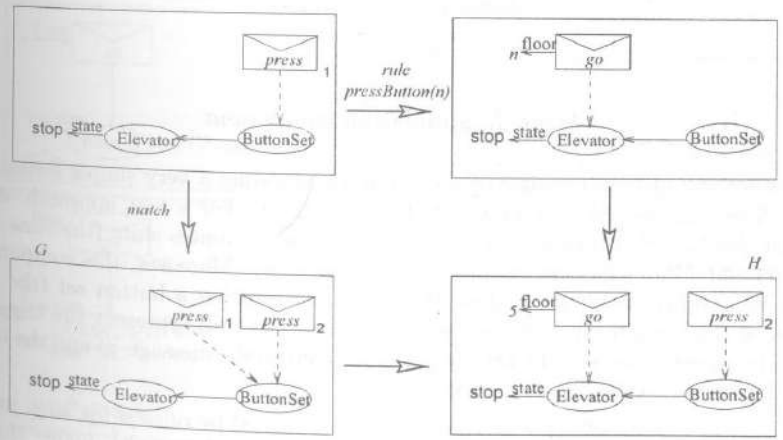

Figure 4: **Application of a rule**

Figure 4 shows a rule application. This rule models the event of a button being pressed. In reaction to the message *press* the elevator receives a message to go to floor

n. The floor number n is given as rule parameter. It is bound to a value by the user / simulation controller when the rule is applied. In this example, n is bound to the value 5. Note that there are two possible matches for this rule because there are two messages *press* in graph G. We have indicated the chosen match by corresponding indices.

3 The PLATUS Simulation Environment

In the PLATUS environment [CK98, RC00, CMR00], a simulation model is constructed as a composition of simulation components. The first step to specify a component is to describe its internal structure as a graph. Then one can specify its behavior via graph rules and an initial state. This gives raise to the *BehaviorGG* of Figure 5. As we are constructing a simulation model, we now have to specify how this component shall be animated. But, in order to obtain a flexible model, we will not specify the animation concretely within the specification of the component, but only the messages shall be sent to the animation module and when these messages must be sent. Actually, this specification of animation is an enrichment of the *BehaviorGG* by adding new messages (and maybe attributes, if necessary). This resulting graph grammar will be called *AbsAnimGG* because it is a very abstract description of the animation. The concrete choice of graphical layout for this specification will be done when choosing an animation module to interpret the corresponding messages. An analogous reasoning gives raise to the *AbsStatGG*, a graph grammar that describes abstractly the procedures for statistical data collection/analysis used in this component. The interface will be discussed in Section 5 .

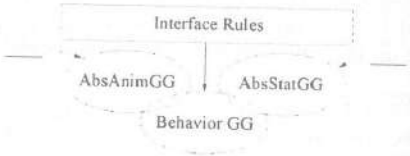

Figure 5: **Simulation component**

Now we will give an example of a component modeling a very simple elevator. In Figure 6 we can see the type graph of this component. For a first approach, do not consider the dashed elements. A node *Elevator* has as attributes *state* (the state of the elevator) and *floor* (the current floor the elevator is in). Moreover, this component is known by another component called *ButtonSet*, representing a button set (the specification of this component will not be shown here). We will represent the triggers of actions by message nodes. The elevator shall respond to the message *go* and the button set shall respond to the message *press*.

The reaction of the elevator to a message *go* is expressed by rules *goUp*, *stop*, *goDown* depicted in Figure 7. Rule *goUp* specifies that, if the target floor (n) is higher than the current floor (m) then the elevator goes one floor upwards, sets its state to up and sends a message to itself in order to trigger the application of the same rule, once more, until it reaches the desired floor (the triggering message is always consumed by the application of the rule). The other rules behave analogously.

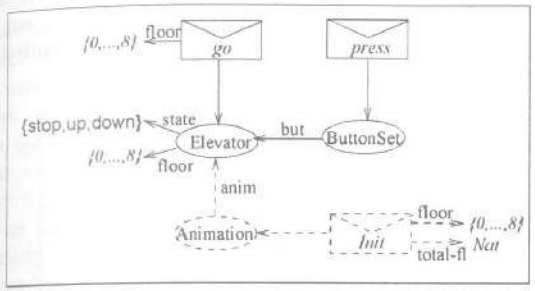

Figure 6: **Elevator type graph**

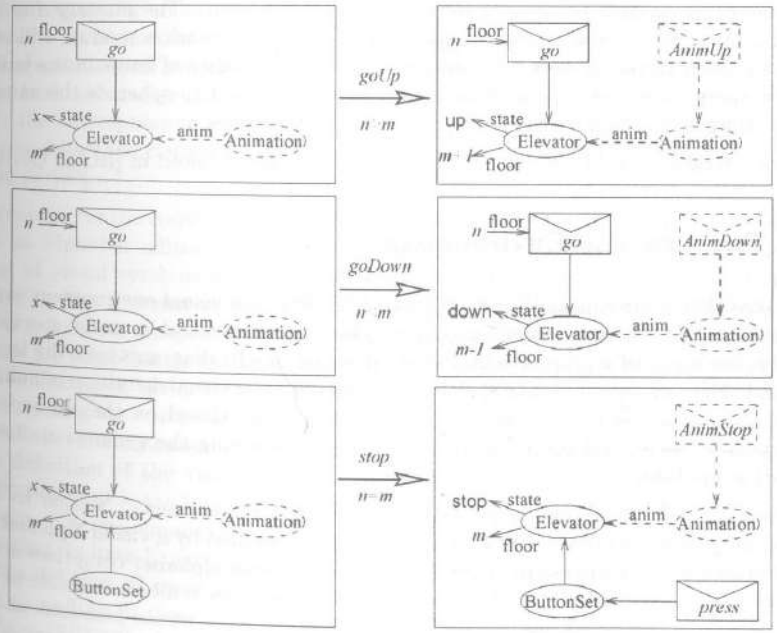

Figure 7: **Elevator rules**

The initial graph *Ini* is shown in Figure 8. It specifies that, in the beginning, the elevator must be in the ground floor (0), in the state stop. Moreover, the button set has a trigger *press*. Note that the specification of the button set includes the rule *pressButton*(*n*), as shown in Figure 4. When this rule is applied, a message *go* is generated on the elevator starting the elevator movements.

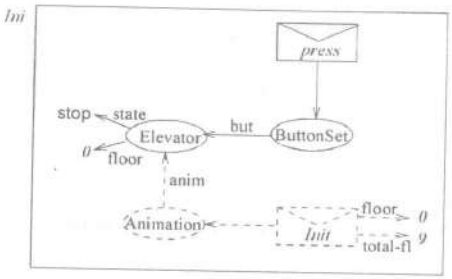

Figure 8: **Elevator initial graph**

The dashed messages and components in Figures 6 – 8 represent graph objects that are not present in the *BehaviorGG* of the elevator but only in its *AbsAnimGG*. For example, in the rule *goUp* it is specified that, besides the behavior described above, a message *AnimUp* shall be sent to the animation component. The message *Init* in the initial graph *Ini* describes that, in the beginning, the animation module will receive this message with parameters 9 (representing the total number of floors in the building) and 0 (representing the current floor). This message is used to generate the animation picture that is modified by the application of the other rules.

More details about PLATUS and its architecture can be found in [RC00, CMR00].

4 The GENGED Environment

The GENGED environment [Bar00, BNS00, BTMS99] is a visual environment supporting the visual definition of visual languages. The result is a visual language specification which is the input of a graphic editor that allows us to edit diagrams over the language specified. Here we show how GENGED can be used for the visual definition of animation modules; we consider each module as a visual language. Based on these descriptions, in Section 5 we extend the GENGED approach by allowing the generation of several animation modules.

The GENGED environment implements its underlying concepts. Similar to formal textual languages, in GENGED, a visual language is specified by a visual alphabet and a visual grammar. In contrast to textual languages, a visual alphabet comprises not only the symbols of a language but additionally the links between symbols, because in visual languages we have to deal with two-dimensional expressions whose positions and sizes must be taken into account. As we model visual grammars by graph grammars, a visual grammar of a visual language consists of a start graph and a set of graph grammar rules. Like graph grammar rules, also visual grammar rules may be context-sensitive such that GENGED supports the visual specification of a broad spectrum of visual languages.

In GENGED we distinguish between two syntactical description levels, namely, the abstract syntax level and the concrete syntax level of visual languages. The abstract syntax describes the language's elements (e.g. the names for symbols and links), whereas the concrete syntax describes their layout. The layout is given by the description of graphical objects and graphical constraints between the objects. In general, graphical constraints express relations between graphical objects concerning their positions and sizes. We use the notion of *constraint satisfaction problem* [DvB97] in order to define a set of constraint variables correlating with certain object properties (positions and sizes), and equations (graphical constraints) over the constraint variables. The problem is to find a solution for all the constraints. For example, we can specify that a circle is to be placed inside a box by a constraint. Then, if the box is moved, a constraint satisfaction problem has to be solved in order to calculate the new position of the circle. In literature, a lot of solving algorithms can be found; we use the constraint solving algorithm (and the corresponding constraint solver system) described in [Gri96] that is able to find the most adequate solution.

The visual specification of visual languages (VLs) supported by GENGED is an application of algebraic graph transformation [Bar00]. This is similar to the concepts of typed graph grammars introduced in Section 2 but more complex with respect to the graphical objects used for the layout, and the correlation of object properties and constraint variables occurring in constraint satisfaction problems. Formally, a visual alphabet is given by an algebraic graph structure signature and a constraint satisfaction problem. This means that a visual alphabet is interpreted as a type graph. Additional access operations on graphics define the constraint variables. All instances (VL diagrams, both sides of grammar rules) are typed over the alphabet and moreover, they satisfy the corresponding constraint satisfaction problems. For the derivation of diagrams we follow the concepts mentioned in Section 2 together with graphical constraint solving [Gri96].

According to the constituents of a VL specification, the GENGED environment comprises an alphabet editor and a grammar editor. The alphabet editor supports the editing of visual symbols for both, the abstract syntax and the concrete syntax, respectively. The abstract syntax of visual symbols is represented by node types. The node type attributes represent the concrete syntax of visual symbols. Once the visual symbols are established, the links between these symbols can be defined. On the abstract syntax level, the definition of links establish arc types between the symbol types. For connecting the corresponding symbol layouts, the user can define several graphical constraints. The alphabet is the input of the generic grammar editor supporting the visual definition of the visual grammar over the visual alphabet. The result is a VL specification that is the input in the intended generic graphic editor for a specific visual language. The grammar rules are used as the language-specific edit commands. This means that we have language-generating rules but also language-manipulating rules for, e.g. the deletion of some symbols. The transformation of diagrams in the grammar editor as well as in the graphic editor is supported by the graph transformation system AGG [TER99]; the constraint satisfaction problems are satisfied by the graphical constraint solver PARCON [Gri96].

In the following we concentrate on our example, namely the specification of an animation view for an elevator. In Figure 9, several visual symbols and visual links of the

corresponding visual alphabet are shown. There we have two lexical symbols *Elevator* and *Animation*. These symbols are equipped with attribute symbols *TotalFloors*, *Floor* and *Message*, respectively. The *Elevator* is illustrated as part of a building showing several floors. The total number of floors depends on a concrete number modeled by *TotalFloors*. The *Animation* is illustrated by an elevator cabin which is intended to move between the floors of the building. The *Floor* attribute is used to indicate the current floor of the animation view. On the layout level (concrete syntax) the current *Floor* number is not visible but is represented by drawing the elevator cabin in the corresponding floor of the building. The *Message* attribute symbol is used to indicate the movement of the elevator between the floors. The layouts of the corresponding enumeration data type are given on the concrete syntax level. We do not visualize neither the *init* nor the *stop* state but the directions *up* and *down* (illustrated by arrows up and down).

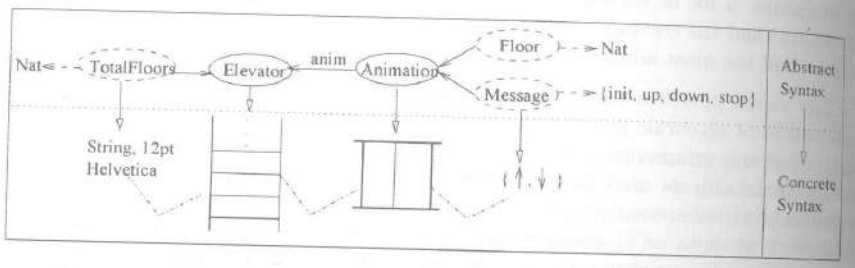

Figure 9: **Visual alphabet for one animation view of an elevator**

Not only the symbols as shown in Figure 9 are important for the animation view. Additionally we expect some messages triggering the movement of the elevator. These messages concern mainly the abstract syntax level, i.e., the animation of the visual language. Therefore, we extend our alphabet by some symbols and links in Figure 10 for the messages. These messages have influence on the animation which is modeled by rules later on but the messages should not be visualized.

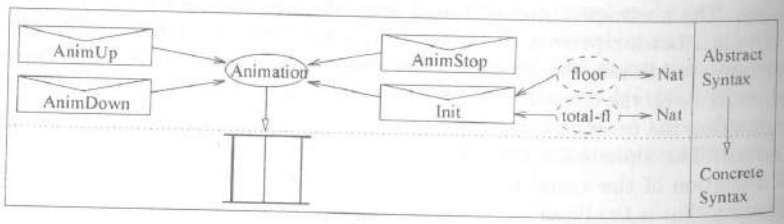

Figure 10: **Extended visual alphabet**

Now we have defined the visual alphabet of our visual language for the elevator example. Based on this alphabet, we define some rules according to the messages we expect to receive and which trigger the animation. Figure 11 illustrates the rule for the initialization of the animation view. In the left-hand side we expect the *Init* message holding the amount of available floors in the building given by the variable t and the

floor where the elevator currently is (f). From this message, the animation view is built up in the right-hand side of the rule.

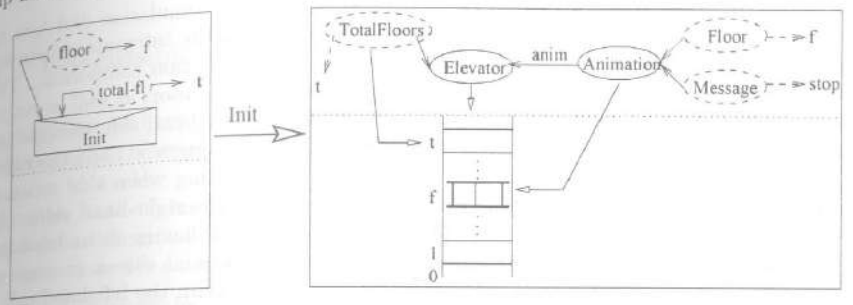

Figure 11: **Rule supporting the initialization**

Similar to the *Init* rule, the rule for the *AnimationUp* movement is modeled and shown in Figure 12. In its left-hand side we require the existence of an *Elevator* and *Animation* symbol which are connected due to the alphabet. Furthermore, we expect a corresponding *AnimationUp* message. This message triggers an upwards movement of the elevator indicated by the floor $f + 1$ in the rule's right-hand side. On the concrete syntax level, the up movement is illustrated by an arrow beside the elevator.

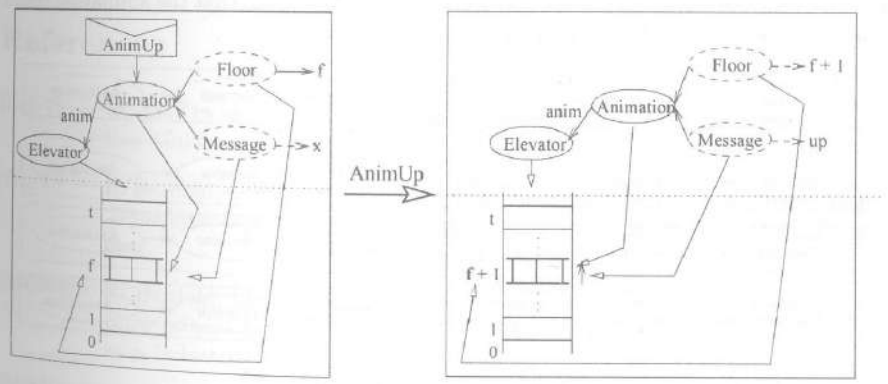

Figure 12: **Rule supporting the up movement**

The remaining rules, e.g., for stopping the elevator and for the downwards movement, are similar to that of Figures 11 and 12 and will be therefore omitted here.

Within this section we have shown how to use GENGED to describe one specific animation view for the elevator example, giving raise to one animation module. The idea is to construct a library of such modules that can be reused by many components.

5 Animation of Simulation Models

Up to now we have defined how to specify a component of a simulation and how to construct animation modules. In this section we will discuss the integration between these two. In Figure 5, we have the basic structure of a simulation component. The interface of such a component is composed by rules that specify import as well as export actions. An imported action is represented by a rule whose left-hand side specifies the message that triggers that action (sent to the corresponding component) and the right-hand side specifies the message that this component is expecting when this action is completed (if this is the case, if not, there is no message in the right-hand side). An example of an imported action of the elevator would be a rule having in its left-hand side a message *press* sent to a button set and on the right-hand side a message *go* sent to the elevator. Export messages are analogous, just that in the left-hand side a message handled by this component is shown, and in the right-hand side the message sent in return (if it exists). As the rest of the component, the interface is organized into behavior, animation and statistics interface, whereas the animation and statistics components do not have any export rules (because the animation and statistics modules do not depend on the behavior module). In the following we will stick to the animation interface.

In our example, the corresponding interface are shown in Figure 13. Rule *up-int* states that the elevator may send a message *AnimUp* to its animation module and will expect nothing in return (no message as an answer). Actually, it would be quite unusual to expect answers from such messages because this would mean that the animation could affect the behavior of the elevator, what is not desirable.

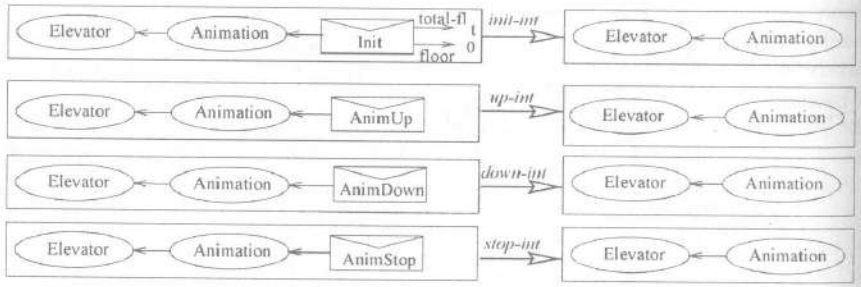

Figure 13: **The interface**

Each animation module also have an (export) interface composed by rules, describing which are the messages that are treated by this module. For our example, this interface would consist of four rules that are the inverses of the four rules of the animation interface of the component (inverse rule means to exchange left- and right-hand side).

When constructing a simulation model, the user must choose the components that he/she will use, and also the animation and statistics module for each of these components. A consistency check must be performed to guarantee that the chosen animation/statistics modules are able to treat the desired messages. Note that this consistency check is the same that is done for importing/exporting other components (behavior).

Thus, we have achieved a uniform way to deal with behavior/animation/statistics within the simulation model.

6 Conclusion

In this paper we have presented an approach to construct animation modules using the tool GENGED. The idea is to use this tool to construct a library of animation modules that can be used to visualize the behavior of a simulation model. Here we used graph grammars as the basis formalism to describe both, the animation and the behavior of the system to be simulated. The integration of the simulation model with the animation models is done via an interface consisting of rules.

Here we have settled the first concepts that shall serve as a basis for the corresponding implementation of interfaces. But still a lot of work has to be done in order to obtain better animation strategies. One is the construction of animation for complex models, like the ones composed of various components. In this case, we may need to give a graphical layout for the composition operators (to see, for example, two elevators within the same building). Another improvement would be to consider richer interfaces, containing, for example, some information about relationships among messages (like, message *AnimUp* is opposite to *AnimDown*) This could be useful to assure that the visualization is in a way compatible with the semantics of the model being simulated.

References

[Bal97] O. Balci et al, *The Visual Simulation Environment*, 11th European Simulation Multiconference, SCS, 1997, pp. 61–68.

[Bar00] R. Bardohl. *Visual Definition of Visual Languages based on Algebraic Graph Transformation.* PhD thesis, Technische Universität Berlin, 2000. To appear.

[BNS00] R. Bardohl, M. Niemann, and M. Schwarze. GENGED – *A Development Environment for Visual Languages.* Application of Graph Transformations with Industrial Relevance, LNCS 1779, pages 233–240. Springer, 2000.

[BRMB96] B. T. Barcio, S. Ramaswamy, R. Macfadzean, and K. Barber, *Object Oriented Analysis, Modeling and Simulation of a National Air Defense System*, Simulation (1996).

[BTMS99] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. *Application of Graph Transformation to Visual Languages.* G. Rozenberg, editor, Handbook of Graph Grammars and Computing by Graph Transformaitons, Volume 2: Applications, pages 103–180. World Scientific, Singapore, 1999.

[CK98] B. Copstein and L. Korff, *Specifying Simulation Models Using Graph Grammars*, ESS'98 10th European Simulation Symposium, SCS, 1998, pp. 60–64.

[CMR96] A. Corradini, U. Montanari, and F. Rossi. *Graph Processes*. Fundamentae Informatica, vol. 26, no. 3-4, 1996, pp. 241–265.

[CMR00] B. Copstein, M. Móra, and L. Ribeiro. *An Environment for Formal Modeling and Simulation for Graph Grammars*, 33rd Annual Simulation Symposium, 2000.

[Cop97] B. Copstein, *SIMOO : Plataforma orientada a objetos para simulação discreta multi-paradigma*, Ph.D. thesis, Federal University of Rio Grande do Sul, 1997.

[DvB97] R. Dechter and P van Beek. *Local and Global Relational Consistency*. Theoretical Computer Science, 173:283–308, 1997.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach*. In [Roz97].

[Ehr79] H. Ehrig. *Introduction to the Algebraic Theory of Graph Grammars*. V. Claus, H. Ehrig, and G. Rozenberg, editors, 1st Graph Grammar Workshop, LNCS 73, pages 1–69. Springer, 1979.

[Gri96] P. Griebel. *ParCon - Paralleles Lösen von grafischen Constraints*. PhD thesis, Paderborn University, February 1996.

[KN96] T. Kamigaki and N. Nakamura, *An Object Oriented Visual Model-building and Simulation System for FMS Control*, Simulation (1996).

[Kor95] M. Korff. *Generalized Graph Structures with Applications to Concurrent Object-Oriented Systems*, Ph.D. thesis, Technische Universität Berlin, 1995.

[Low93] M. Löwe. *Algebraic Approach to Single Pushout Graph Transformation*. Theoretical Computer Science, vol 109, 1993, pp. 181–224.

[RC00] L. Ribeiro and B. Copstein. *Compositional Construction of Simulation Models using Graph Grammars*, International Workshop and Symposium AGTIVE - Applications of Graph Transformation with Industrial Relevance, LNCS 1779, pages 87–94. Springer, 2000.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformaitons, Volume 1: Foundations*. World Scientific, Singapore, 1997.

[TER99] G. Taentzer, C. Ermel, and M. Rudolf. *The AGG Approach: Language and Tool Environment*. G. Rozenberg, editor, Handbook of Graph Grammars and Computing by Graph Transformaitons, Volume 2: Applications, Languages and Tools, pages 551–604. World Scientific, 1999.