Animated Simulation of Integrated UML Behavioral Models based on Graph Transformation*

Claudia Ermel
Fac. Electrical Eng. and Comp. Science
Technical University of Berlin
Franklinstr. 28, 10587 Berlin, Germany
lieske@cs.tu-berlin.de

Karsten Hölscher, Sabine Kuske, Paul Ziemann
Dept. for Math. and Comp. Science
University of Bremen
P.O. Box 330 440, 28334 Bremen, Germany
{hoelscher,kuske,ziemann}@informatik.uni-bremen.de

Abstract

This paper shows how integrated UML models combining class, object, use-case, collaboration and state diagrams can be animated in a domain-specific layout. The presented approach is based on graph transformation, i.e. UML model diagrams are translated to a graph transformation system and the behavior of the integrated model is simulated by applications of graph transformation rules. For model validation, users may prefer to see the behavior of selected model aspects as scenarios presented in the layout of the application domain. We propose to integrate animation views with the model's graph transformation system. A prototypical validation system has been implemented recently supporting the automatic translation of a UML model into a graph transformation system, and the interactive execution and simulation of the model behavior. We sketch the tool interconnection to GenGED, a visual language environment which allows to enrich graph transformation systems for model simulation by features for animation.

1 Introduction

In the last decade the Unified Modeling Language [2,24] has become more and more popular to visualize and document different aspects of software systems. Unfortunately, most of the UML diagram types were introduced without a formal semantics that maps them to a mathematically well-understood domain. Furthermore, different UML diagrams represent different system aspects, but their coupling is not precisely defined, i.e. even if one had a formal semantics of every employed diagram type it would not be always clear in which way the diagrams and their semantics interact.

Graphs and graph transformation have been employed successfully for both the specification of a well-founded formal semantics of different UML diagrams [7, 12, 19, 22, 32] as well as the definition of an integrated formal semantics of a UML model consisting of a set of UML diagrams [14, 20, 34, 35]. The latter has the advantage that the behavior of the entire system described by a UML model can be simulated so that system states are graphs (representing enriched object diagrams) and simulation steps are modeled by the application of graph transformation rules. In other words we obtain for every model specified by UML diagrams a graph transformation system (called simulation system) which simulates the model's behavior via the iterated application of graph transformation rules.

Graph transformation proved to be an adequate means for a formal description of UML models for various reasons. On the one hand, the area of graph transformation is theoretically well-founded and thoroughly studied [6, 28]. On the other hand, UML diagrams can be represented in a straightforward way as graphs, and system evolution can be naturally described and executed via the application of graph transformation rules. Moreover, there exist some well-developed graph transformation tools [5].

However, for validation purposes this integrating graph-transformation-based approach for the formalization of a UML semantics is not always adequate. System states are visualized in simulation runs as graphs which may become rather complex. This is due to the fact that the simulation system is generated automatically, which necessarily involves auxiliary constructs and yields more complex rules than a hand-written specification would contain.

Therefore, in this paper we extend the integrated graph transformation-based UML semantics by *animation views* which allow to define model-specific scenario animations in the layout of the application domain (cf. also [8]). We call the simulation steps of a behavioral model *animation steps* when the states before and after a step are shown in

*Research partially supported by the EC Research Training Network SegraVis (Syntactic and Semantic Integration of Visual Modeling Techniques)

the animation view. Fig. 1 sketches the relation between UML behavioral models and their animation views in different application domains. An integrated UML behavioral

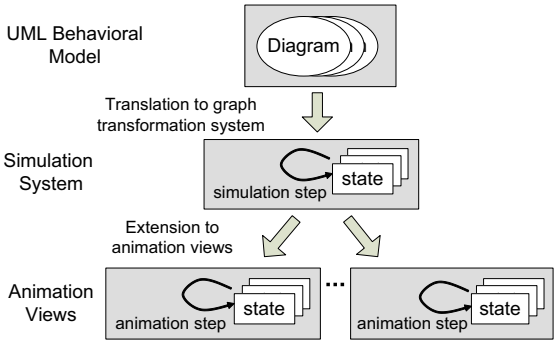

Figure 1. Animation views for UML models

model (consisting of several diagrams) is translated automatically to a simulation system, which is then extended on the basis of a *view transformation system* (another graph transformation system) [9]. The view transformation system realizes a consistent mapping from simulation steps to animation steps in the respective animation view.

It is worth noting that our notion of *animation* goes beyond the notion of *specification animation* in the literature, which means to generate an executable prototype from a formal specification [23]. More precisely, in the context of UML, *animation* means for example the following:

- Generating "filmstrips" from the UML model (snapshot sequences of object diagrams over a given class diagram [25]). This approach is closely related to our simulation system with the difference that we use graph transformation rules instead of OCL constraints.

- Enriching UML behavioral diagrams by graphical means for highlighting process steps, such as colored message arcs in sequence diagrams or colored activity rectangles in activity diagrams (see e.g. the animation add-in for Microsoft Visio [30]).

- Enhancing the UML diagram syntax, e.g. by three-dimensional layout to make UML diagrams more readable. Gogolla et al. [13, 26] use three-dimensional blocks for classes in class diagrams and move message balls between objects of a sequence diagram.

We prefer to regard the execution of a prototype generated from a behavioral model, such as our graph transformation system, as *simulation*, even if the syntax is enhanced by highlighting or 3D-features.

Animation in this paper is also based on the simulation prototype but differs from simulation in three respects: First, simulation presents the states of the system in the abstract, formal syntax of the modeling language (e.g. as object diagrams or graphs), whereas animation uses a domain-specific layout (an *animation view*), which is visually closer to the modeled system and hides the underlying formalism. Second, simulation shows model state transitions as discrete steps, whereas animation shows a continuously changing scenario in a movie-like fashion. At last, simulation shows complete states whereas animation may abstract from implementational details of the model that are not important to understand functional behavior.

We advocate that this way of animation simplifies the early detection of inconsistencies and possible missing requirements in the model which cannot always be found by simulation and analysis only.

A prototypical system implementing the translation of a UML model into a graph transformation system was developed in [29]. Moreover, the visual environment GenGED [11] allows to enrich graph transformation systems with animation features and to export scenario animations in SVG format [33]. In this paper we also sketch the tool interconnection of both, resulting in a graph-transformation based platform for animating UML behavioral models.

For reasons of space limitations we do not introduce the approach in a general way but illustrate it by a running example modeling a client-server system. The system basically consists of drive-through restaurants and clients. In the animation view, the model behavior is visualized in the layout of cars queueing in front of a drive-through restaurant, giving orders and being served.

The paper is organized as follows. In Section 2, the UML model of the example is presented. Section 3 introduces its integrated semantics as a graph transformation system, and Section 4 shows how an animation view can be built on top of such a system. In Section 5 we give a short overview of the tools supporting our approach. The paper ends with some concluding remarks.

2 A Sample UML Model

In general, UML models are composed of several UML diagrams such as use-case, class, object, state, collaboration, sequence, and activity diagrams (see [2, 24]). The interplay of UML diagrams – as used in our integration approach – is depicted in Fig. 2. An integrated UML model consists of one class diagram and one top-level use-case diagram containing the use cases of the simulation which are refined by collaboration diagrams. For each class, there can be a state machine diagram describing in which order the operations of the class can be executed. Object diagrams which instantiate the class diagram model possible states of the system.

To illustrate the connection between the different UML

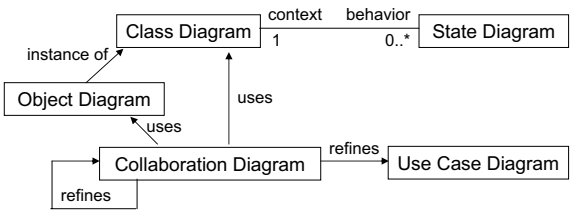

Figure 2. Central UML modeling concepts

modeling concepts, we sketch a sample UML model of a client-server system which consists of an arbitrary number of drive-through restaurants each of which has a (possibly empty) queue of hungry clients that are served one after the other. There may also exist some more idle people who are not yet visiting a drive-through.

The user of the drive-through simulation may select idle people to be hungry and drive-throughs to start serving clients. Hence, the *use-case diagram* of our simulation system consists of two use cases, namely callClientToEat and startDriveThrough. Both are refined by a set of collaboration diagrams, one of which is presented below.

The *class diagram* is depicted in Fig. 3. It consists of the four classes Client, DriveThrough, Meal, and Order. Clients may be associated with a drive-through via a Visit-association (in this case they are hungry). Clients may submit orders and eat meals that are served by drive-throughs. Drive-throughs and clients can perform a series of operations the most important of which are shown in Fig. 3. The names of meals and orders are given by attributes.

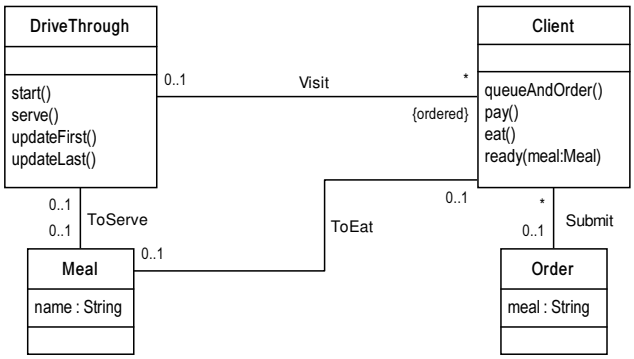

Figure 3. Class diagram for drive-through model

Instances of class diagrams are *object diagrams* consisting of an arbitray number of objects for every class, and links for every association so that the multiplicity requirements of the associations are satisfied.

Fig. 4 presents the *protocol state machine* for the class Client. Initially, a client is in the state idle. In this state the operation queueAndOrder can be executed which also changes the state of the client from idle to waiting. In the state waiting the client can pay and change its current state to hasPaid. After paying the client can eat and be idle again. Operations which do not occur in the state machine diagram can be executed in every state. (By the collaboration diagrams of the model it is guaranteed that the operation ready is executed only between the operations pay and eat.)

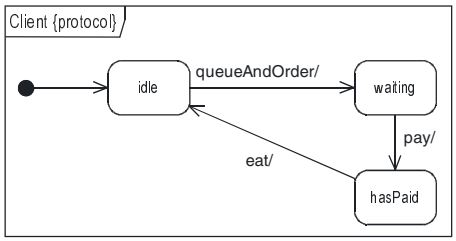

Figure 4. State machine diagram Client

Operations of classes can be described with *collaboration diagrams*. The collaboration diagram in Fig. 5 specifies the operation serve in which a drive-through d serves a meal m ordered by the first client c of its queue. The operation creates the meal-object m (1.1). It inserts a ToServe-link between d and m (1.2). The attribute name of m is set to the meal-attribute of the order submitted by c (1.3). Drive-through d sends the message ready(meal) to c (1.4) with the effect that the Submit-link attached to c is deleted (1.4.1) and a ToEat-link between c and m is created (1.4.2). At last, the ToServe-link between d and m is deleted (1.5).

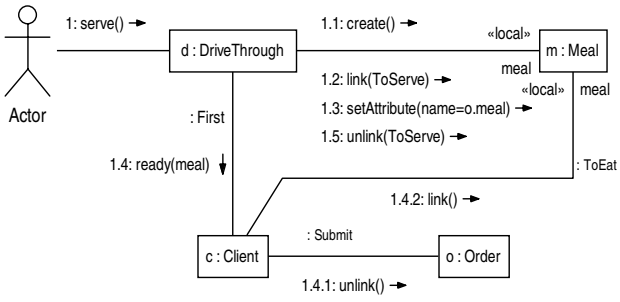

Figure 5. Collaboration diagram for serve()

It is worth noting that the main operations modeled in the collaboration diagram are the creation and deletion of links or objects or the sending of messages. Graph transformation systems can model such operations in a straightforward way. In the next section we sketch how UML models can be automatically translated into graph transformation systems in order to obtain a proper semantics of UML models.

3 Integrated UML Semantics based on Graph Transformation

The underlying idea of defining the operational semantics of a system specified via UML diagrams is to represent system states as (enriched) object diagrams (formalized as graphs) and system evolution steps as transformations of such object diagrams (formalized as graph transformation).

Up to now, the integrated semantics of UML takes into account use-case diagrams, class and object diagrams, state machine diagrams, and sequence and collaboration diagrams. The automatic integration of all these diagrams is a graph transformation system mainly consisting of a set of graph transformation rules and a graph representing the initial system state. The operational semantics of the system consists of all system states that can be reached from the initial state via the iterated application of graph transformation rules.

3.1 Graph Transformation

A *graph transformation system* consists of an initial graph and a set of graph transformation rules which rewrite parts of graphs when applied. We use the algebraic graph model for attributed, directed and labeled graphs and their transformations [21], where a *graph transformation rule* consists of two graphs L and R, called left and right-hand side which may share a common part. A rule can be applied to a graph if there exists an image of the left-hand side in this graph. The *application* of a rule deletes the image of the left-hand side up to the common part, removes all dangling edges, and replaces it by a copy of the right-hand side up to the common part. It is also possible to use negative application conditions [16] which only allow a rule application if some context specified in the negative application condition does not occur in the current graph. Boolean OCL expressions may also be used as application conditions meaning that a rule can only be applied if the OCL expression is evaluated to true. Variables representing attribute values in the usual way can be used in both sides of a rule.

Fig. 6 presents an example of a graph transformation rule. Both sides of the rule contain three common nodes representing an object of class DriveThrough, an object of class Client and an object of class Order. The application of the rule to a graph inserts an additional node (an object of class Meal) into the graph it is applied to.

This rule realizes the effect of operation 1.1 of the collaboration diagram in Fig. 5. There the execution of operation 1.1 create is specified to create a new Meal object.

In general, it is possible to associate types with the nodes and edges of the graphs that are transformed by a graph transformation system. This can be done by specifying a so-called type graph [3].

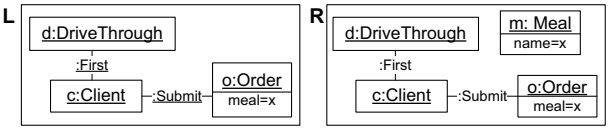

Figure 6. Rule for the operation serve():1.1 create(Meal)

3.2 The Integrated Semantics

As stated before, the integrated semantics of a UML model is a graph transformation system in which system states are modified via the application of graph transformation rules. A system state is a kind of object diagram representing the objects that are alive in the system state.

Fig. 7 shows an excerpt of a system state of our drive-through model. It consists of two clients (o1 and o2), together with their current states, a drive-through (o3), and an order (o4). Both clients are visiting the drive-through, o1 being the first and o2 being the second and last client in the queue. The order has been submitted by the first client.

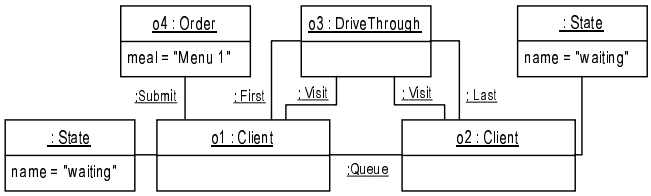

Figure 7. A system state of the drive-through model

Technically, the classes, their attributes and operations, the associations, the initial object states and a node for each use case are also included in every system state. For reasons of space limitations and clarity they are omitted here.

An initial state of a UML model consists of all objects and links (together with the corresponding classes, associations, initial states, etc.) that are alive in the beginning of the system simulation. (Fig. 7 does not show an initial state because the clients are not in their initial state idle.)

Each operation of a class and the use cases are translated into a set of graph transformation rules according to the specifications given by the collaboration and state machine diagrams. Technically, the control flow of operation execution is modeled by means of process nodes. These nodes fall into two main categories: complex and atomic ones. *Complex* process nodes represent operations that are composed of suboperations as specified in the collaboration diagrams. *Atomic* process nodes represent predefined, basic operations which specify the creation of objects, the in-

sertion of links between them, the setting of their attribute values, etc.

When a state machine diagram is specified for an object, the execution of an operation may not be defined to take place in a certain object state. In this case, a corresponding process node is added to the system state, but it can never get active. In this way, an operation call will be ignored if the called object is in a "forbidden" state. Our approach supports basic state machine diagrams (protocol state machines) without advanced features like nested states.

Should the model be incomplete (e.g. a class operation without a specification in a collaboration diagram), no rules are generated for the incomplete part. Thus the system execution will get stuck, due to the lack of adequate rules (in case of a non-specified operation there will be no rules to properly execute and terminate it, thus succeeding operations may never be executed).

The rule presented in Fig. 6 summarizes the effect of a set of four more detailed rules that are applied in a special order. These detailed rules would be applied to the complete system state graph, of which Fig. 7 shows an excerpt.

Another example for a rule corresponding to the operation 1.4 in the collaboration diagram for serve() in Fig. 5 is the rule shown in Fig. 8. As specified by the operations 1.4.1 and 1.4.2 of the collaboration diagram, the Submit link between a Client object and an Order object is deleted, and a ToEat link between the Client object and the Meal object is created under the condition that the name attribute of the Meal object corresponds to the meal attribute of the Order object. Again, this is a summary of a set of more detailed rules realizing the unlinking and linking in several steps, taking into account also the link ends.

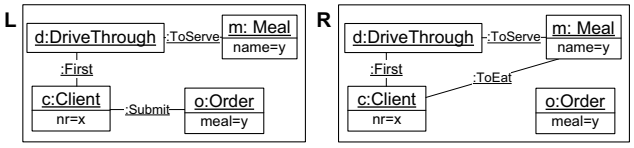

Figure 8. Rule for the operation serve():1.4 ready(meal)

Summarizing, for each operation of a class specified in a collaboration diagram we can automatically generate a set of graph transformation rules the application order of which is determined by so-called process nodes. Furthermore, the state machine diagrams determine in which object states the operations can be executed. Finally, all parts of the class diagram and the use cases are reflected in the system states. Hence, the graph transformation system is obtained from the use-case, class, state, and collaboration diagrams of the model.

4 Animation Views for Integrated UML Behavioral Models

Despite the benefits of graph-transformation based simulation, the behavior of a complex formal model may not always be comprehensible to users. This is due to the fact that many details had to be integrated in the simulation system in order to be able to perform a formal analysis. Moreover, due to the automatic generation of the simulation system, several auxiliary structures are generated which simplify the generation process and schedule the order of rule applications, but make it more difficult to understand immediately the behavior of the resulting generated system (analogously to program code which is generated automatically).

Therefore, instead of simulating the model behavior in its whole complexity, we propose the use of *animation views* and *view transformation* for well-founded model animation based on graph transformation. Thus, in a systematic way, additional graphical symbols for the animation of model behavior are added to the behavioral model, such that on the one hand users can easily understand and validate the model behavior, and on the other hand, the animation view does not change the semantics of the modeled system.

An animation view presents the model states in the layout of a specific application domain. The simulation steps of a behavioral model are called animation steps if the states before and after a step are shown in the animation view.

The view transformation is realized in three steps: at first, the visual language definition for the simulation system (a type graph for extended object diagrams) is extended by symbol types and relations describing the graphical means used in the respective animation view for the domain-specific representation of model states. Secondly, the view transformation rules are defined over this extended type graph, and applied to the initial graph of the simulation system, generating animation view symbols and linking them to all objects which shall be animated. Thirdly, the same view transformation rules are applied to the simulation rules such that afterwards the state transitions also visualize the animation view symbols. The resulting transformed simulation rules are called *animation rules*, and together with the transformed initial graph we obtain an *animation system*. A *scenario animation* then is defined as a derivation sequence applying the animation rules beginning with the initial state of the animation system. Moreover, by adding continuous animation operations to the animation rules the resulting scenario animations are not only discrete-event steps but can show the model behavior in a movie-like fashion. Consequently, requirements and scenarios can be validated in one or more animation views.

In contrast to software visualization approaches relying on the observer pattern [10] for decoupling the model and its visualization, in our graph transformation-based ap-

proach, the basic visualization information is kept in the graph transformation rules in close relation to the model. This allows to prove in a formal way that the model behavior is preserved in the visualization, by defining *views* as graph transformation systems, and arguing about properties of view morphisms [9]. Despite this close relation of a model and its animation view, the decoupling of both is realized, as the animation view is generated by a separate graph transformation system. Thus, if animation features need to be changed, the underlying model itself is not touched, but only the view transformation rules have to be adapted and applied again to the model, resulting in a changed animation view for the model.

4.1 An Animation View for the Drive-Through Restaurant

For the example of our drive-through restaurant we choose an animation view where the clients are visualized as cars queueing in front of a restaurant building. Their orders are shown as bubbles inscribed by the order attribute meal, e.g. "Meal 1". Animated actions are the entering of the drive-through, ordering, being served, eating and leaving the queue. The animated actions (and their graphical representations) are coupled to certain graph transformation rules which evoke the corresponding operations. Other graph transformation rules (preparing the operation processing or checking conditions) are not relevant for the animation view. They will be still applied but their effect will not be visualized in a scenario animation.

The Extended Type Graph

Fig. 9 shows the extended type graph for our animation view. At the top we have the type graph for extended object diagrams (the states of the simulation system), and at the bottom we have the new symbol types needed for the animation view. Both parts (the old and the new type graph) are linked by an arc connecting the type Object and its visualization type Building. This top-level link is sufficient to define how a specific drive-through object, its clients and their orders are visualized because all animation view symbols are connected to a certain building object or to a car which in turn is connected to a building.

The extended type graph contains not only information how symbols of a certain symbol type should be visualized (the symbol icons), but also layout constraints, i.e. relations concerning the size and positions of the symbol type graphics. For example, there have to be layout constraints defining that cars are always positioned on the road, that a car should be placed in front of its successor in the queue, that an order bubble points to its car, and that a served

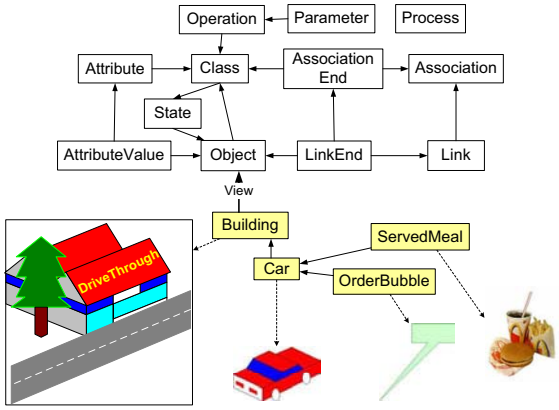

Figure 9. The extended type graph for the animation view

meal is positioned near the car which submitted the order. Formally, these layout constraints are defined as a graphical constraint satisfaction problem (a set of equations over positions and sizes of graphics) which has to be satisfied by each diagram which is an instance of the type graph.

The View Transformation Rules

Fig. 10 shows the abstract syntax (without layout information) of the view transformation rules which are used to add the new animation view symbols in a consistent way to the initial graph and to the simulation rules of the drive-through simulation system. Rule application is done on the abstract syntax level only, and the layout of the derived graph is computed according to the layout constraints in the extended type graph. The first rule, initDriveThrough, adds the top-level icon for the animation view and links it to a drive-through object. The other rules require a link from a client object, either connecting it to a drive-through, or to an order or to a meal, and add the respective animation view symbols and their links. Note that for all rules the negative application condition is the same graph as the right-hand side, such that each rule can be applied exactly once to each of the links of a client. In our approach, we do not delete the original graphs but only add animation view symbols. The visual language tool GENGED (see Section 5.2) allows to mark parts of a type graph as invisible, such that in the animation we show derivation sequences visualizing only the graph objects belonging to the animation view.

Fig. 11 shows a sample animation rule which is obtained by applying the view transformation rules to the simulation rule in Fig. 8.

In Fig. 12 we finally present a few snapshots from a sample derivation sequence, visualized in the animation view.

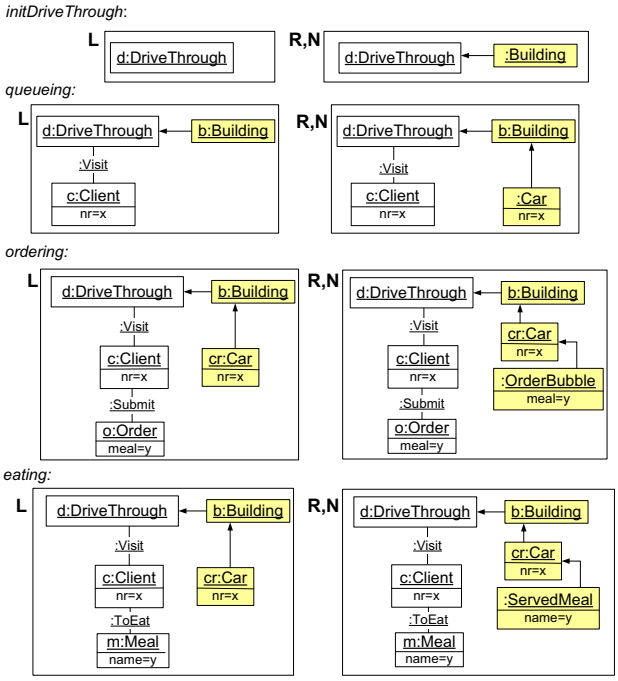

Figure 10. View transformation rules for the drive-through model

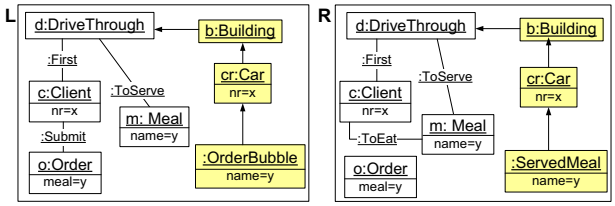

Figure 11. Rule serve():1.4 ready(meal) **translated to an animation rule**

We apply the graph transformation rules modeling the operations for order(), pay(), serve() and eat(). The first graph shows two cars queueing in front of the drive-through building (idle cars are not visualized in the animation view). Then the first client in the queue submits an order (visualized by an order bubble, inscribed by the name of the ordered meal), the client pays (not visualized), the meal is prepared and served (visualized by a meal icon placed next to the car), and finally, the meal is eaten and the client leaves the drive-through.

Using GENGED, continuous movement of cars is specified by adding a *linear-move* operation to the animation rules modeling a client to enter or to leave the queue (rules queue() and eat()).

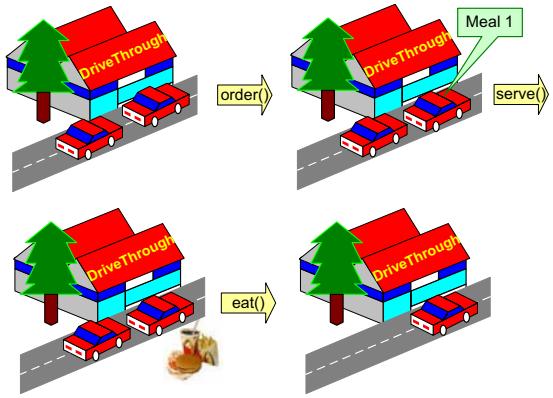

Figure 12. A derivation sequence visualized in the animation view

5 Implementation

5.1 UGT: Simulation by Graph Transformation

In order to support the automatic generation of rules from the given UML model and to facilitate a stepwise execution of the simulation system, the prototypic tool UGT (*UML to Graph Transformation*) has recently been developed [29]. UGT reads a model specification from a given text file and automatically generates the graph transformation rules as introduced in [34,35]. The initial graph is computed from the input model plus an object diagram specified by the modeler as initial. Should the model specification be incomplete, the automatic rule generation will fail. In case of success the user may run the obtained simulation system by choosing one of the specified use cases to be executed. Internally, when a use-case is chosen, the corresponding set of graph transformation rules is applied stepwise to the graph representing the current system state. This can be done automatically by selecting the *step* command in the tool, or interactively by clicking on a waiting process node. In order to check e.g. invariants when executing use cases or processes, it is at all times possible to evaluate OCL expressions in the context of the current system state.

Technically, UGT combines two well established tools in order to realize the functionality described above. The graph transformatorial part of UGT is realized by the graph transformation tool AGG [31] and an extension of the validation tool USE [27] is utilized for the evaluation of OCL expressions.

5.2 GenGED: Animation based on Graph Transformation

Tool support for the animation of visual behavior models has been realized using GENGED [1, 11], a tool for gener-

ating visual modeling environments. In GENGED, an alphabet editor supports the definition of the language vocabulary (alphabet) as type graph and the layout of alphabet symbols by graphical constraints. For the layout computation, GENGED uses the graphical constraint solver PARCON [15] which supports the definition of complex graphical constraints, e.g. to declare minimal or maximal distances of graphical objects, or to define relations between objects like *above, inside* or *at_border*. Such complex constraints are decomposed by PARCON into basic constraints, i.e. equations over positions and sizes of graphical objects. A valid layout for a given model is a set of values for the layout positions and sizes of all graphical objects belonging to the model such that the constraint equations are all evaluated to true, i.e. the layout constraints defined by the alphabet are satisfied.

A visual grammar editor allows to define different kinds of grammars based on the alphabet, e.g. for syntax-directed editing, parsing and/or simulation. Alphabet and grammars configure a specific visual language environment, including an editor for the specified language. The simulation rules are used for validating the model behavior, where the underlying graph transformations are performed by AGG [31]. Moreover, they are the basis for the definition of an application-specific *animation view* as demonstrated in this paper. Related tools for generating visual modeling and simulation environments based on graph transformation are e.g. DiaGen [17] and AToM^3 [4]. In DiaGen, the focus lies on the generation of sophisticated visual editors, and the main applications of AToM^3 are the simulation of discrete and continuous semi-formal models by graph transformation. Both tools do not provide support for user-defined animation in a different layout.

In GENGED, scenario animations can be exported to the SVG format [33] and viewed by an external SVG viewer which shows continuous state changes according to the defined animation operations. Due to the generic and modular definition of syntax, behavior and animation for behavior models, the GENGED approach reduces the amount of work to realize a domain-specific animation of a system's behavior. The tool integration of UGT and GENGED is based on AGG which is the underlying graph transformation engine for both tools. Thus the graph transformation system which has been generated from the UML model specification by UGT can be imported as simulation grammar and used for the animation specification by GENGED.

6 Conclusion

In this paper we have combined two approaches for modeling and validating system behavior based on graph transformation. The first approach defines a translation from central UML language features into graph transformation systems to obtain a formal, integrated UML semantics (using the tool UGT). The second approach extends graph transformation systems modeling behavior, by animation views that enable users to run scenario animations based on the graph transformation rules, but visualized in a domain-specific layout (using the tool GENGED). The combination of both approaches has been demonstrated by a model of a drive-through restaurant.

The main benefit of the combination of both approaches are, on the one hand, a precise UML semantics obtained by the translation into a graph transformation system, and, on the other hand, the flexible visualization of the simulation steps in the animation view. The precise semantics allows to execute use cases and to check invariants by evaluating OCL constraints at arbitrary execution states. The animation view allows to hide auxiliary, technical nodes in the graph that have proved necessary for the automatic translation but rather confusing for a user trying to simulate the system behavior Thus, the animation supports an intuitive validation of user requirements, in addition to the formal checking of invariants.

Future work is planned to cover still more selected language features from UML, especially considering the UML 2.0. Moreover, in complex cases the integration of various UML diagrams may lead to large graph transformation systems which are difficult to handle and to understand. Therefore, for practical use, structuring concepts for graph transformation (see e.g. [18]) should be incorporated in the presented approach, and also implemented in the tool AGG, which is the underlying graph transformation engine for both UGT and GENGED. Work is in progress to implement type graphs with inheritance and multiplicities as underlying language model in AGG, which should make it easier to go the step from a meta model description (a class diagram) to the corresponding type graph for a UML based visual language.

References

[1] R. Bardohl, C. Ermel, and I. Weinhold. GenGED – a visual definition tool for visual modeling environments. In J. Pfaltz and M. Nagl, editors, *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, 2003.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[3] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–265, 1996.

[4] J. de Lara and H. Vangheluwe. ATOM^3: A tool for multi-formalism modelling and meta-modelling. In R. Kutsche and H. Weber, editors, *Proc. Fundamental Approaches to Software Engineering (FASE'02)*, volume 2306 of *LNCS*, pages 174 – 188. Springer, 2002.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by*

Graph Transformation, Vol. 2: Applications, Languages and Tools. World Scientific, Singapore, 1999.

[6] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific, Singapore, 1999.

[7] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000 – The Unified Modeling Language. Advancing the Standard*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[8] C. Ermel and R. Bardohl. Scenario animation for visual behavior models: A generic approach. *Software and System Modeling: Special Section on Graph Transformations and Visual Modeling Techniques*, 3(2):164–177, 2004.

[9] C. Ermel and K. Ehrig. View transformation in visual environments applied to Petri nets. In G. Rozenberg, H. Ehrig, and J. Padberg, editors, *Proc. Workshop on Petri Nets and Graph Transformation (PNGT)*, volume 127(2) of *ENTCS*, pages 61–86. Elsevier, 2005.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] GenGED homepage. http://tfs.cs.tu-berlin.de/genged.

[12] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy et al., editors, *Proc. ICSE'98 Workshop Precise Semantics of Modeling Techniques*, pages 55–72, 1998.

[13] M. Gogolla, O. Radfelder, and M. Richters. Towards three-dimensional representation and animation of UML diagrams. In R. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, volume 1723 of *LNCS*, pages 489–502. Springer, 1999.

[14] M. Gogolla, P. Ziemann, and S. Kuske. Towards an integrated graph based semantics for UML. In P. Bottoni and M. Minas, editors, *Proc. Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2002)*, volume 72/3 of *ENTCS*, 16 pages. Elsevier, 2003.

[15] P. Griebel. *ParCon: Parallel Solving of Graphical Constraints*. PhD thesis, Univ. of Paderborn, Germany, 1996.

[16] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[17] O. Köth and M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In H. Ehrig and G. Taentzer, editors, *Proc. Workshop on Graph Transformation Systems (GraTra2000)*, pages 32–39. Technical Univ. of Berlin, 2000.

[18] H.-J. Kreowski and S. Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11:690–723, 1999.

[19] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *UML 2001 – The Unified Modeling Language. Modeling languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.

[20] S. Kuske, M. Gogolla, H.-J. Kreowski, and R. Kollmann. An integrated semantics for UML class, object and state diagrams based on graph transformation. In M. Butler, L. Petre, and K. Sere, editors, *Proc. 3rd Int. Conf. on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 11–28. Springer, 2002.

[21] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. R. Sleep, R. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting, Theory and Practice*, pages 185–199. Wiley & Sons, Chichester, 1993.

[22] A. Maggiolo-Schettini and A. Peron. A graph rewriting framework for Statecharts semantics. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*, pages 107–121. Springer, 1996.

[23] I. Morrea, J. Siddiqi, R. Hibberd, and G. Buckberry. A toolset to support the construction and animation of formal specifications. *Systems and Software*, 41:147–160, 1998.

[24] Object Management Group. *OMG Unified Modeling Language Specification, version 1.5*, http://www.omg.org, 2003.

[25] I. Oliver and S. Kent. Validation of object-oriented models using animation. In *Proc. EuroMicro'99, Milan, Italy*, 1999.

[26] O. Radfelder and M. Gogolla. On better understanding UML diagrams through interactive three-dimensional visualization and animation. In V. D. Gesu, S. Levialdi, and L. Tarantino, editors, *Proc. Advanced Visual Interfaces (AVI 2000)*, pages 292–295. ACM Press, New York, 2000.

[27] M. Richters. A UML-based Specification Environment, last revision 2001. http://www.db.informatik.uni-bremen.de/projects/USE.

[28] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, Singapore, 1997.

[29] L. Schaps. Design and implementation of a system executing UML and OCL specifications based on graph transformation (in German). Master's thesis, Univ. of Bremen, 2005.

[30] Sysoft. *Visio Diagram Animation with Amarcos*, http://www.sysoft-fr.com/en/Amarcos, 2004.

[31] G. Taentzer. AGG: A graph transformation environment for system modeling and validation. In T. Margaria, editor, *Proc. Tool Exihibition at Formal Methods'03*, 2003.

[32] D. Varró. A formal semantics of UML Statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 1st Int. Conf. on Graph Transformation*, volume 2505 of *LNCS*, pages 378–392. Springer, 2002.

[33] WWW Consortium (W3C). *Scalable Vector Graphics (SVG) 1.0 Specification*, http://www.w3.org/TR/svg, 2000.

[34] P. Ziemann, K. Hölscher, and M. Gogolla. Coherently Explaining UML Statechart and Collaboration Diagrams by Graph Transformations. In A. Moura and A. Mota, editors, *Proc. Brazilian Symposium on Formal Methods*, volume 130 of *ENTCS*, pages 263–280. Elsevier, 2005.

[35] P. Ziemann, K. Hölscher, and M. Gogolla. From UML models to graph transformation systems. In M. Minas, editor, *Proc. Workshop on Visual Languages and Formal Methods*, volume 127(4) of *ENTCS*, pages 17–33. Elsevier, 2005.