Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Visual Languages and Computing
17 (2006) 78–105

Journal of
Visual Languages
& Computing

www.elsevier.com/locate/jvlc

On translating UML models into graph transformation systems[☆]

Karsten Hölscher[*], Paul Ziemann, Martin Gogolla

Department of Computer Science, University of Bremen, Bremen, Germany

Accepted 9 November 2005

Abstract

In this paper we present a concept of a rigorous approach that provides a formal semantics for a fundamental subset of UML. This semantics is derived by translating a given UML model into a graph transformation system, allowing modelers to actually execute their UML model. The graph transformation system comprises graph transformation rules and a working graph which represents the current state of the modeled system. In order to support UML models which use OCL, we introduce a specific graph transformation approach that incorporates full OCL in the common UML fashion. The considered UML subset is defined by means of a metamodel similar to the UML 1.5 metamodel. The concept of a system state that represents the state of the system at a specific point in time during execution is likewise introduced by means of a metamodel. The simulated system run is performed by applying graph transformation rules on the working graph. The approach has been implemented in a research prototype which allows the modeler to execute the specified model and to validate the basic aspects of the model in an early software development phase.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Graph transformation; UML semantics; Validation; CASE tool

[☆]Research partially supported by the EC Research Training Network SegraVis (Syntactic and Semantic Integration of Visual Modeling Techniques) and the project Abstract Implementation of and Documentation with UML (UML-AID) funded by the German Research Foundation (DFG).

[*]Corresponding author.

E-mail addresses: hoelscher@informatik.uni-bremen.de (K. Hölscher), ziemann@informatik.uni-bremen.de (P. Ziemann), gogolla@informatik.uni-bremen.de (M. Gogolla).

1045-926X/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2005.11.001

1. Introduction

The Unified Modeling Language (UML) has recently become a standard for the modeling of object-oriented software systems. It comprises a set of different diagram types, each of them describing particular aspects of software artifacts. The syntax of these diagrams is described by means of a metamodel in [1], denoted as class diagrams. Since the class diagram itself is defined in a cyclic way by the metamodel, the metamodel definition of UML diagrams can only be considered semi-formal. Furthermore the semantics of UML components is only expressed in natural language. In order to overcome the limitations of a purely graphical notation, the UML has been enhanced by the textual Object Constraint Language (OCL). The OCL is also semi-formally defined in [1]. A formal syntax and semantics for UML class diagrams as well as OCL has been introduced in [2], which is also included in the accepted OCL 2.0 submission to the OMG [3].

This work presents a concept for obtaining a formal semantics not only for class diagrams but for further basic diagram types (use case, object, statechart and interaction diagrams) belonging to the UML standard 1.5. We stick to UML 1.5 but UML 2.0 likewise includes the UML concepts covered by us, albeit some details and the naming have been changed in some cases. In particular, UML 1.5 collaboration diagrams are called communication diagrams in UML 2.0. Graph transformation (cf. [4–6]) is employed as the formal foundation of this new integrating semantics.

Our approach provides a framework for an automatic translation of a UML model into a graph transformation system. The UML model may comprise a subset of the diagram types mentioned above and may furthermore include OCL expressions. The graph transformation system consists of graph transformation rules and a so-called working graph, hence called system state graph. This graph represents the state of the modeled system at a given point of time. The changes of the system state during an execution of the model are simulated by the application of graph transformation rules on the system state graph. In this way a stepwise execution of the model can be simulated. As no formal semantics is given for the UML, the effects of the model execution rely on a number of assumptions, especially regarding the integration of the mentioned diagram types and their use in practice.

We have enhanced the graph transformation foundation with OCL expressions. These expressions navigate the current system state and can be used as application conditions, which determine whether a certain rule may or may not be applied. OCL expressions can also be used to calculate new attribute values in the right-hand side of graph transformation rules. Additionally, OCL is used as query language for inspecting the current system state. The use of OCL as a textual notation also leads to the benefit of more compact graphs in most cases.

The representation of a UML model as a graph transformation system is used here to validate the system before actually implementing it. Employing graph transformation for the simulation of the modeled system has the benefit of a visualized system run. The simulation allows modelers to compare the behavior of the system with their expectations. Given a system state α, they can easily gain an understanding of the actions that are possible in this state. Furthermore our concept also supports goal-oriented tests regarding the question whether a given state ω is reachable from state α (though this is generally undecidable as it is equivalent to the word problem for any kind of language). The approach also enables the modeler to check what states can be derived from the state α. The integration of OCL allows for the checking of OCL invariants during system state evolutions. The modeler may test whether an invariant that holds in state α still holds in a derived system state ω.

Currently, a prototypic validation system is implemented for our approach which generates the graph transformation rules for a given model and allows to interactively execute and visualize the modeled system. This prototype can be used by a modeler in an early stage of a software development process in order to acquire additional insight into the newly designed system.

A full formal and elaborated description of our concepts can be found in the Ph.D. thesis [7], where the integrated system specification and the translation into a graph transformation system as well as the implemented prototype UGT are discussed: in the central part of that work, the translation of an integrated UML model into a graph transformation system is described in detail. The concept of system states, which are represented by graphs and transformed by graph transformation rules, is defined. The thesis explains how to derive the initial system state, which is the system state at the beginning of a system run, from the given object and statechart diagrams. The graph transformation rules are described subsequently. The work describes the rule for initiating use cases, which is independent of the UML model. Also independent of the model are the rules performing predefined operations like creating objects or setting attributes. The paper describes rules executing object operations, i.e., operations the user has declared in a class diagram and specified by an interaction diagram. Use cases are treated as operations that do not belong to a class (use case operations), and therefore the rules realizing use cases are constructed in the same way. Nodes that are not needed anymore are deleted from the system state by the garbage collection rule. Finally, the work clarifies the way class generalization and thus inheritance of attributes and operations are covered. For all details, we again refer to the original work [7].

The structure of the rest of this paper is as follows. In the next section we present some related work, followed by a section in which the concepts of graph transformation we employ are introduced. The covered UML features of the model are presented and explained using a simple example in Section 4. Section 5 deals with the detailed description of the system state concept. An overview of the translation of the model into a graph transformation system is presented in Section 6 mostly by example, followed by a brief introduction to the fundamental architecture of the prototypic implementation. The paper closes with concluding remarks in Section 7.

2. Related work

Since UML lacks a formal mathematical foundation, several works can be found that address this issue. Different formalisms are employed to provide a formal semantics for parts of UML. In [8] a translation of UML into a partial Object-Z specification is presented. The work in [9–11] discusses the specification of UML semantics on the metamodel level. Ref. [12] presents a formal semantics of UML activities based on Petri-nets. Streams are employed as semantic domain of a UML model in [13]. The works [14,15] focus on high level semantics based on temporal and deontic logic, respectively. The xUML approach presented in [16] defines a subset of UML for rigorous object-oriented modeling and provides an operational semantics for supported diagram types.

There are also several other works aiming at defining a semantics for parts of UML using graph transformation. In [17], an integrated semantics is given for a large part of UML.

However, interaction diagrams and OCL are not considered. Their approach is extended with interaction diagrams on instance level in [18]. Operations are still specified by single rules, that is, all operations have to be atomic. More efforts exist considering isolated parts of UML. In [19], collaborations are translated into transformation rules, where collaborations are interpreted as visual queries using pattern matching. A formal semantics for UML statecharts is presented for example in [20,21]. The Fujaba tool suite [22] supports graphical object-oriented software design and automatic code generation from story diagrams. These diagrams combine behavioral UML diagrams and additional features. Additional approaches for consistency analysis of UML models can be found. In [23], given UML real time models are refined using graph transformation rules and their consistency is checked in the semantic domain of CSP. Ref. [24] studies questions concerning the realization and the semantics of UML packages in connection with a graph-based tool. Ref. [25] addresses the consistency analysis between UML class and sequence diagrams based on graph transformation. Parts of the UML semantics have also been defined with Abstract State Machines (ASMs). Work includes the UML semantics definition for single diagram like statechart [26,27] and activity diagrams [28] and the study of special aspects like constraints [29] or liveness [30]. Furthermore, ASMs have been used for the validation of UML models [31] and the implementation of a UML virtual machine [32]. However, a comprehensive integrating treatment of these different works seems to be missing.

All these works have in common, that they mostly address isolated parts of UML. We are not aware of an approach handling a collection of UML diagrams as presented in this work together with the integration of OCL. In particular the incorporation of use cases is new. In [33] use cases are described precisely by so-called operation schemata including OCL pre- and postconditions but the connection to other UML diagrams is left open.

The main benefit of our approach is (A) the integrated coverage of a substantial part of the UML, (B) a minimal impedance mismatch between the original UML model and our semantical domain, in particular the system state and (C) the possibility to validate and test on the model level. As the system evolves, our system state graph changes accordingly, always in tight closeness to the actual visual UML model. The system state may be understood as a UML object diagram. Analogously to the system state, our rules, e.g., for statechart transitions, bear strong similarities to the original UML descriptions. The closeness between UML model and the semantical domain allows to give better feedback to the modeler because the modified system state can again be understood as a UML description. Our approach also makes fewer assumptions regarding the semantics of the model as, for instance, other approaches and code generators have to make. In order to improve the quality of software, the application of code generators is an accepted means in the industrial software development. As direct tests of the model are not yet supported, the generated code is debugged. Should errors occur during tests, currently the generated code is changed. But in our approach, it is possible to directly perform tests on the level of the model without having to deal with generated code. Our approach also allows for an easier handling of future extensions and changes of the model, since only the model itself and not the generated code has to be changed.

3. Graph transformation

Graphs are a frequently used means to visualize complex information, like the structure of computer networks or database designs. Graph transformation allows local changes on such a graph by applying graph transformation rules. A very obvious application domain for graph transformation is the field of visual modeling, like UML. A model may be transformed into another model of the same visual language (e.g. refinement or refactoring) or into a model of a different visual language (cf. MDA). A graph transformation system may also serve as a semantic domain, as is the case in our approach. Various well-studied graph transformation approaches can be found in the literature.

Basically a graph consists of a set of vertices, a set of edges, and mappings s, t that assign a source and a target vertex to every edge. Furthermore vertices and edges are mapped into a given alphabet, providing edge and vertex labels. The label alphabet includes the invisible label, which is used for vertices or edges whenever they are supposed to carry no label.

Since we want to represent more sophisticated structures where objects may also contain attributes, we enhance this graph model with a concept for vertex attributation. Analogously to the presentation in [34], data values are represented as special data value nodes, all of which are present in the attributed graph. The elements of the combined set of data value nodes and "usual" vertices will be called nodes. The source, target and label mappings then apply to the set of all nodes, where the data value nodes are labeled with their respective type. In our approach the data value nodes are the elements of the mathematical sets $\mathbb{Z}, \mathbb{R}$, $\{\text{true}, \text{false}\}$, A^* (for a given alphabet A) representing the OCL types *Integer*, *Real*, *Boolean*, and *String*, respectively. An attribute of a vertex is then represented by an edge from that vertex to the corresponding data value node. This edge is labeled with the name of the attribute. Since the carrier sets are usually infinite, data value nodes are omitted in pictures unless they have incoming edges. In this case we call the data value nodes *visible*.

In figures we employ a simplified notation of vertex-attributed graphs by depicting them in a UML-like fashion. Vertices are depicted as rectangles with two compartments. In the upper compartment the vertex identifier and its label can be found, separated by a colon. Here the identifier is used as a mere reference for easier description of a graph. In the lower compartment the attribute names together with their concrete values are depicted one per line. Fig. 1 shows an attributed graph with two vertices and three data value nodes on the left-hand side, and the corresponding simplified version of that graph on the right-hand side.

In this paper a graph transformation rule consists of three graphs L, K, R, called left-hand side, common part, and right-hand side, respectively. In order to increase the flexibility of rules, the data value nodes of the left-hand side are extended by a set X of variables. Thus it is possible to specify variable values instead of concrete constant values for attributes in the left-hand side of a rule.

Attribute data value nodes of the right-hand side are somewhat more complex in order to allow for attribute value computation. We assume that every concrete data value is represented syntactically by a constant symbol. Let $B = (B_i)$ be the family of these constant symbols, indexed by the basic types listed above. Furthermore let $\Sigma_B(X)$ be the set of *expressions* over B and X which is defined as usual: an expression is either a constant $c \in B$, or a variable $x \in X$, or it is of the form $\omega(t_1, t_2, \ldots t_n)$ with ω being an operation symbol and the arguments $t_i \in \Sigma_B(X)$ being expressions of suitable types. In our case we use the usual operations, e.g. arithmetic operations on numbers, sign manipulation and the like. The data value nodes of the right-hand side are then all elements of $\Sigma_B(X)$. In figures the operations are written in infix notation, i.e. $+(5, x)$ becomes $5 + x$.

The two rule sides L and R are connected by a common part K, which is a subgraph of both L and R. K is a subgraph of L if the nodes and edges of K are subsets of the nodes and

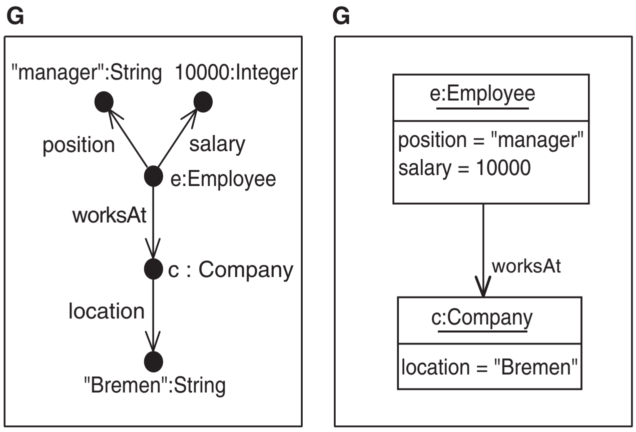


Fig. 1. An attributed graph in two different notations.

edges of L, respectively, and the nodes and edges coincide in their respective label, source and target mappings. Informally speaking a rule is applied to a given graph G (called *host graph*) by finding a situation in G that is specified by L. Then the part corresponding to $L - K$ is deleted from G and $R - K$ is glued to G. In figures depicting rules, the identifier of vertices is used in order to indicate parts of the gluing graph K, i.e. a vertex with the same identifier in the left-hand side and the right-hand side is an element that has to be preserved. Elements without identifiers are either deleted from (in the left-hand side) or added to the graph (in the right-hand side).

In order to find a situation specified by L in a host graph G, a so-called *match* is needed, i.e. a structure-preserving mapping (called *graph morphism*) of L into G. A graph morphism maps the nodes and edges of one graph to the nodes and edges of the other graph such that labels and source and target nodes are preserved (ignoring isolated data value nodes). For our purposes we demand an injective match, i.e. equivalent images require equivalent preimages. Since no variable set is present in the host graph, variable value nodes from X may be mapped to any data value node in the host graph, provided that the structural properties are preserved. Fig. 2 shows a rule with a variable *sal* in the left-hand side and a match of L in a host graph G. The match determines a variable binding for the variable data value nodes in L; for instance, $m(sal) = 10\,000$ in Fig. 2. The same variable may be used more than once in the left-hand side via several edges leading to the corresponding variable data node. In figures, variables are printed in italics.

A rule $r = (L, K, R)$ is then applied to G by deleting $m(L - K)$ from G. This is done by removing the vertices and the edges, together with possible dangling edges, i.e. edges with source or target in $m(L - K)$. This yields an intermediate graph Z. If R contains visible data value nodes from $\Sigma_B(X)$, their expressions are evaluated using the variable binding m. If the expression is of the form x with $x \in X$, the edge leading to x is deleted and a new one with the same label leading to $m(x)$ is created. If the expression is of the form $\omega(t_1, t_2, \ldots, t_n)$, the operation and parameters are interpreted in the usual way (in case of $t_i \in X$ using $m(t_i)$) and the edge leading to that value node is replaced with an edge (again with the same label) leading to the result of the evaluated expression. This evaluation yields an instance R' of the right-hand side which has only visible data value nodes from the basic

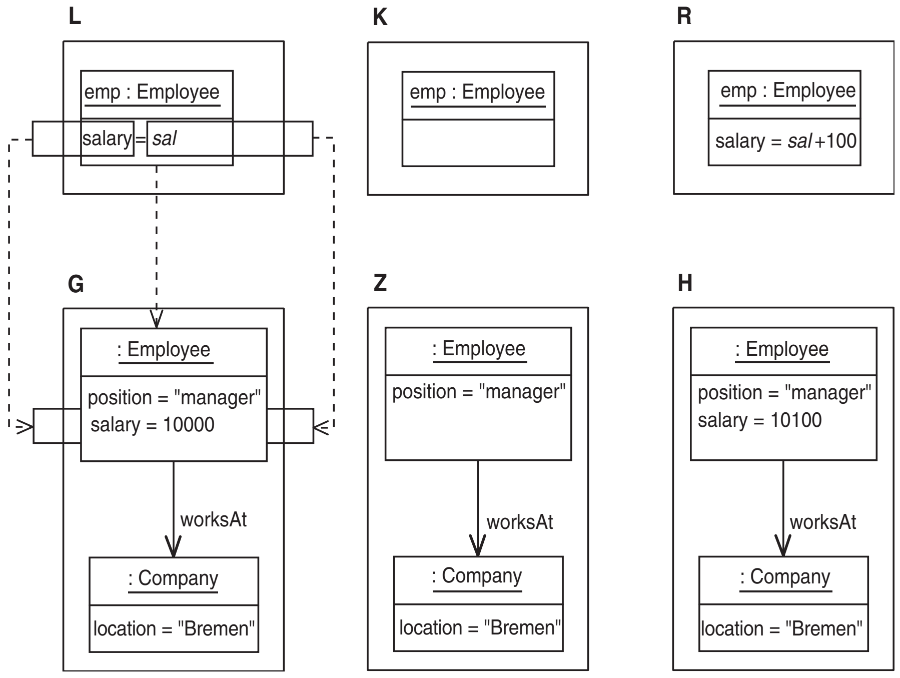


Fig. 2. A rule (L, K, R), a match $L \rightarrow G$ and the application to G.

sets. Now $R' - K$ is glued to Z. This is done by adding all vertices and edges from $R' - K$ to Z, possibly gluing new edges to already present nodes, i.e. if $s(e) \in K$ for an edge e of $R' - K$, then e is connected to $m(s(e))$ (and analogously for $t(e) \in K$). Fig. 2 shows the application of a rule (L, K, R) by depicting G, the intermediate graph Z and the result graph H after successful rule application. The edge labeled `salary` is not part of K, thus it is deleted in Z and added to H as a new edge present in R but not in $R' - K$. The data value node `10 000` becomes invisible in Z, and the data value node `10 100` becomes visible in H.

Besides specifying a desired situation in the left-hand side of a rule, it is sometimes useful to specify a situation in the host graph that is not wanted. This is realized by a *negative application condition* (NAC). An NAC is a graph that extends parts of the left-hand side, i.e. a (possibly even empty) subgraph of L is a subgraph of such a graph NAC. If the match of L in the host graph G can be extended to a match of NAC in G, then the rule cannot be applied. Fig. 3 shows the rule from Fig. 2 together with an additional NAC that prevents the rule from application if the salary to be increased is exactly 100 000.

In our approach, we also employ a simpler form of the so-called *transformation units* [35,36]. In our case a transformation unit comprises a set of local rules and a control condition. To apply a transformation unit to a given graph, the new graph is derived by applying the local rules according to the control condition. The semantics of the operators used inside control conditions will be explained where they occur in the following sections. We regard the application of a transformation unit as an atomic operation, similar to one rule application, since the intermediate graphs are not of any interest. For our purposes the transformation units are constructed in such a way that either their first rule is not

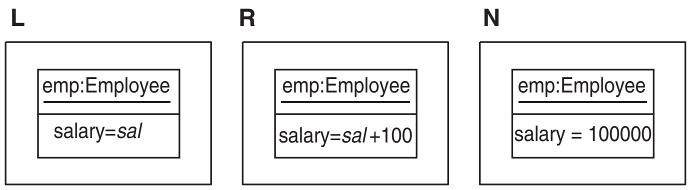


Fig. 3. A graph transformation rule with an NAC.

applicable or the whole unit can be applied successfully.[1] In our approach the semantics of control conditions differs from the original definition in that a failed application of a control condition yields the original (unchanged) graph. In the next section we provide an overview of the UML features that are covered by our approach supported by a very basic sample model.

4. Covered UML features

We cover substantial aspects of the following UML features: use case, class, object, statechart, and interaction diagrams (collaboration and sequence diagrams) and last but not least full OCL.

We support class diagrams for defining the structure, and interaction diagrams for realizing operations declared in the class diagram. An interaction diagram contains a sequence of messages calling either an operation of a class that in turn is realized by an interaction diagram or calling a predefined functionality like creating an object or setting an attribute value.

Use cases are likewise realized by interaction diagrams. A use case resp. its realization states which operations could be called by a user of the eventually implemented system and in which order this is done. Statechart diagrams specify the order in which operations on an object may be executed. The kind of statechart diagrams we support are so-called protocol machines, i.e., statechart diagrams with guards and events used as transition labels. Object diagrams are used to specify the system state to start the evolution with and to represent a part of the current state of the system.

Fig. 4 gives an overview of the connections between the central concepts. We consider one class diagram and one use case diagram. This is no limitation because in our approach multiple class diagrams can be merged to one, and so can use case diagrams. Each class has zero or more operations. A use case is associated with exactly one operation that is not associated with a class. Each operation is realized by an interaction specified in an interaction diagram. An interaction contains messages, which are either predefined (e.g. for creating an object or setting an attribute value) or which call an operation of a class. For each class there can be one state machine specified in a statechart diagram. An object diagram instantiates the class diagram. We illustrate the usage and interplay of the

[1]A property like that would be hard to achieve for transformation units in general, but in the context of our work only a very small subset of all possible transformation unit constructions is needed. Due to this fact and the rule construction scheme in general, the mentioned property can be guaranteed.

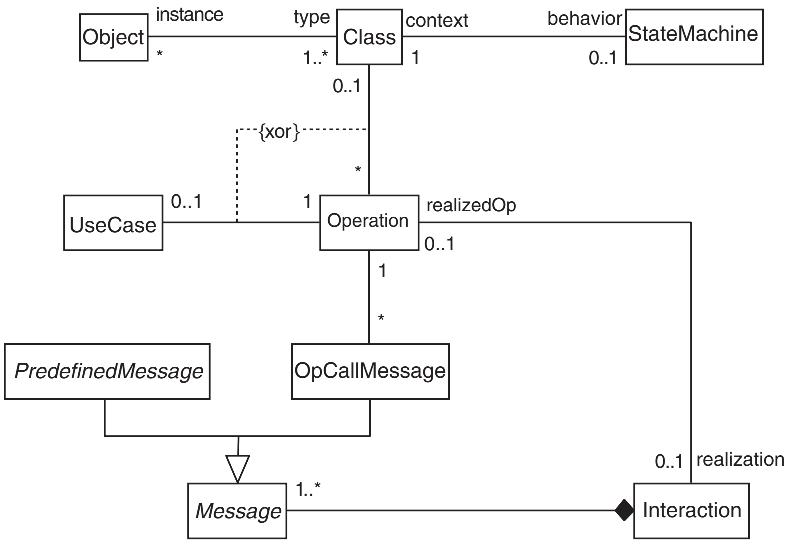


Fig. 4. Connection between central modeling concepts.

diagrams by a representative excerpt of an example UML model of a drive-through restaurant.

The drive-through system consists of clients who enqueue themselves in the queue of a drive-through restaurant, submit orders, pay for meals, and eat. The restaurant produces meals and serves them to the clients.

The class diagram in Fig. 5 defines the structure of our example system. The class DriveThrough represents drive-through restaurants. A drive-through is visited by several clients who await being served. This is specified by the association Visit between DriveThrough and Client. The two additional associations First and Last mark one client as the first one in the queue and another client as the last one. The order in the queue is reflected by the association Queue. Clients can select an order by connecting to an Order object by the association Submit. The drive-through is then expected to produce a Meal object corresponding to the order. This correspondence is managed by the attributes name resp. meal of the two classes. A meal that has been produced and is ready to be served is connected to the drive-through by the association ToServe. A served meal is connected to a client by the association ToEat until it is eaten.

A use case diagram is used to show on a high level the possible interactions of a user with the eventually implemented system. A use case is a sequence of operations the user may call. The use case diagram only shows the names of the use cases; the corresponding sequence is specified in an interaction diagram. In our example there is a use case startDriveThrough which enables the user to control the drive-through, and a use case callClientToEat for "telling" an idle client to queue himself in the queue of a drive-through and submit an order. The user of the drive-through system is modeled as supervisor (see Fig. 6).

Before elaborating on the relatively low-level interaction diagrams, we take a look at statechart diagrams, which are used in a more high-level way in our approach. A statechart diagram specifies the states an object of a certain class can be in, which operations may be

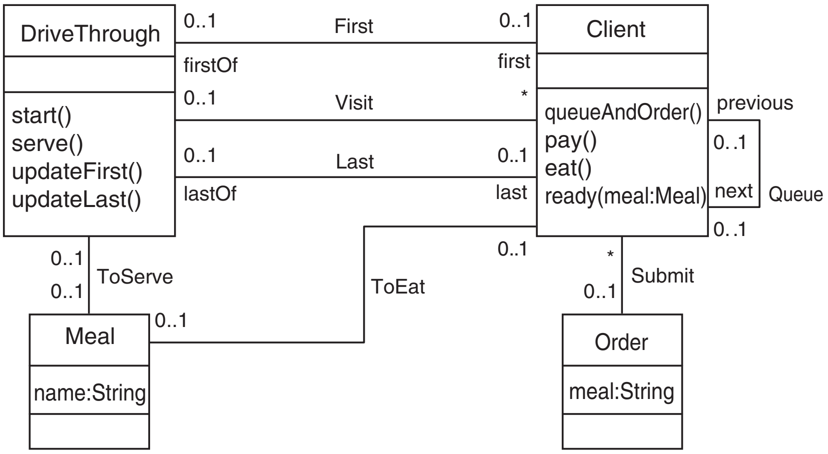


Fig. 5. A class diagram.

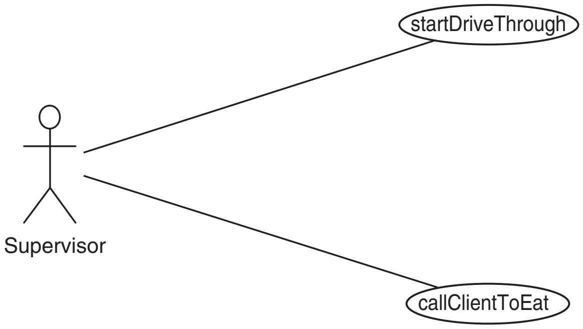


Fig. 6. A use case diagram.

executed in which state, and how the execution of an operation changes the state of the object the operation is running on. We assume that operations which do not occur in the statechart for the corresponding class are allowed to be executed in every state.

The statechart diagram in Fig. 7 specifies the states a client can be in: idle, waiting and hasPaid. The initial state is the state idle, which means that once a client object is created it is in state idle. It is also specified here that executing the operation queueAndOrder is allowed only in state idle and changes the state to waiting. The operation pay then changes the state to hasPaid and the operation eat then changes the state back to idle.

As mentioned above, the operation sequences that constitute use cases are specified in interaction diagrams. In this case we use collaboration diagrams but sequence diagrams could also be used, as explained below.

The collaboration diagram shown in Fig. 8 realizes the use case callClientToEat by specifying the messages the supervisor can send. In this case we have only one OpCallMessage as referred to in the metamodel in Fig. 4. The message (numbered with 1) is sent to a classifier role representing a client and calls its operation queueAndOrder(). Other messages 2, 3, etc. could have followed for a larger use case. The sequence numbers at the beginning of the messages specify the order in which they are sent.

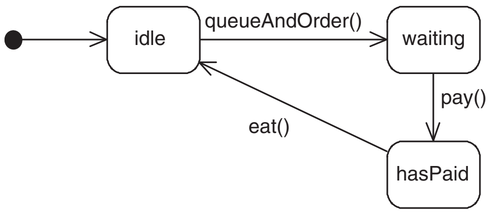


Fig. 7. A statechart diagram for the class Client.

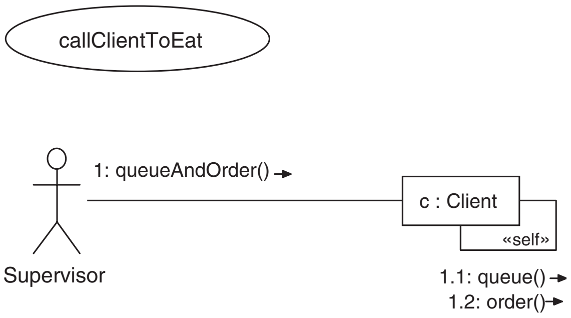


Fig. 8. A collaboration diagram realizing the use case callClientToEat and the operation queueAndOrder of the class Client.

The sequence numbers are nested in different depths to allow the realization of several operations in a single diagram. The present collaboration diagram also specifies the operation queueAndOrder() on Client which is realized by first calling queue() (sequence number 1.1) and then order() (sequence number 1.2) on the object that receives the message.

Fig. 9 shows a more complex collaboration diagram. The operation serve() is called by a supervisor on a drive-through object, which in turn calls several other operations by sending the messages 1.1 to 1.5. Message 1.4 calls the operation ready(meal) on a client, which in turn is realized by the messages 1.4.1 and 1.4.2. The other messages call operations with a predefined effect like creating an object or setting an attribute value.

The nodes in a collaboration diagram are called classifier roles. They represent objects in the system state. The edges are association roles representing links. A message that is sent via an association role that is stereotyped with $\ll$local$\gg$ means that the receiving classifier role is stored in a local variable. The name of the variable is specified by the role name. If the association has no stereotype, the interaction is underspecified. In this case, the message is sent to an arbitrary object of the specified class that is linked to the sender object by a link of the specified association. A future version of the aforementioned tool will alert the modeler of this underspecification and permit them to choose an actual receiver object.

The message 1.2: link(ToServe) specifies that a link instantiating the association ToServe has to be created. Usually a link message needs parameters specifying the concrete objects that have to be linked as well as their association roles (especially if the arity of one association end is greater than one—in this case it is not always obvious which objects have to be linked). In the context of the given model it is not necessary to provide

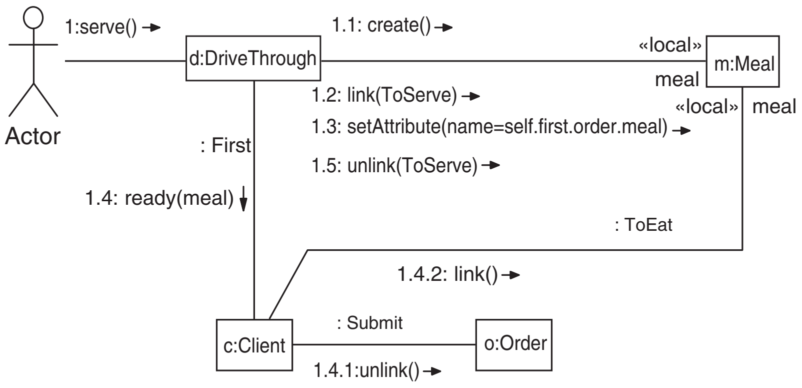


Fig. 9. A collaboration diagram realizing the operations DriveThrough::serve() and Client::ready(meal:Meal).

this information, since ToServe is specified to associate one object of type DriveThrough with an object of type Meal. The actual classifier roles to be linked are in this case the sender d and the receiver m. In case of the link message 1.4.2: link() it is analogously determined which objects will be linked in what way. The usage of an instance of the association ToEat as a channel for the message in this case also determines that the link to be created will instantiate the association ToEat without explicitly providing it as a parameter.

Object diagrams are used in our approach to specify a part of an initial system state, that is the system state the modeler wants to start the system run with. The object diagram in Fig. 10 depicts such an initial system state for the drive-through example. There are one drive-through and four clients. Three of them are visiting the drive-through and have already ordered. The state of the objects is specified by attached notes.

Having specified the relevant parts of the system to be implemented, the modeler may execute the DriveThrough model to check for design flaws. Given the system state corresponding to the initial object diagram in Fig. 10, it could be checked whether client c4 can be inserted into the queue in the right position. It could also be checked whether c4 can be served, or whether the other three clients may submit further orders. A test could also reveal, whether a state is reachable, where all clients have been served according to their orders and the queue is empty. This would be a desired state, at least for the staff of the DriveThrough regarding closing time. An incomplete or contradictory specification can also be revealed in the simulated system run. For instance, in case of a missing link a graph transformation rule relying on this link may never be applicable during the system run. If the modeler forgot to specify the link operation 1.4.2 in Fig. 9, the client would not be able to actually eat the ordered meal, and thus stay in the queue forever. By observing the simulation the modeler should realize this flaw.

In general our approach supports UML models comprising the following syntactical features:

- Use case diagrams with declaration of use cases.
- Class diagrams with classes, *n*-ary associations, and inheritance.
- Object diagrams with objects and links.

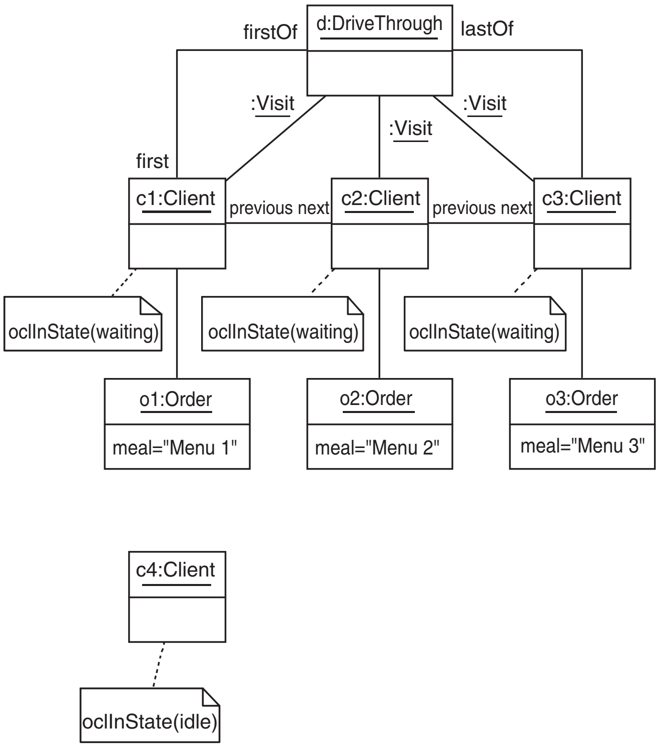


Fig. 10. An object diagram specifying the initial system state.

- Statecharts with simple states labeled with guards and call events (protocol machines).
- Interaction diagrams with ordinary, ≪local≫, and ≪self≫ association roles, sequential, parallel, synchronous and asynchronous messages calling operations on objects or predefined operations.
- OCL expressions in guards of statecharts and interaction diagrams and in arguments of messages.

In our example, we only use synchronous messages, which are visualized by filled solid arrowheads. In contrast to this asynchronous messages are also supported. Before a message is sent it has to wait until the functionality invoked by the preceding synchronous message(s) has finished. Asynchronous predecessors do not have to be finished but they have to be sent.

OCL guards in square brackets in front of messages do not appear in this example but are also supported. Such an OCL condition has to be fulfilled to send the message.

Sequence and collaboration diagrams are based on the same information in the metamodel of UML 1.5 and thus are semantically equivalent (cf. [37], pp. 249–250). It is even possible to convert one diagram type into the other without loss of information [38]. However, in the concrete syntax of sequence diagrams the association roles are not visualized. If the association roles were nevertheless included in a sequence diagram, it could also be used instead of a collaboration diagram in the model.

In the next section we introduce the concept of a system state, which is the essential part of our approach.

5. System states

A system state is a snapshot of the system at some point of time during a system run. It contains attributed objects and links connecting them. So far this graph can be regarded as an object diagram. However, a system state contains two more important concepts: (1) object states, which are attached to objects according to the statechart diagrams, and (2) processes, which represent the actual execution of operations. Briefly, the main concepts of a system state are the following ones:

- Class vertices together with operation and association vertices represent the statical structure.
- Inheritance is represented by connecting the relevant class vertices.
- Object vertices represent existing instances of the connected class vertex.
- Link vertices connected to object vertices represent instances of the connected associations.
- Process vertices represent attached operations in execution.
- State vertices represent the current state of an attached object.
- Local variables are represented by local variable vertices.
- State vertices attached to class vertices determine the initial state for new instances of that class.
- State vertices attached to process vertices determine the state of the corresponding object after the execution of the process has finished.

The abstract syntax of system states is shown in Figs. 11–13 by means of a metamodel. The part of the metamodel shown in Fig. 11 covers the basics of a system state, which are mainly concepts known from UML object diagrams: objects with attribute links and links together with link ends connecting the objects. These elements can be created or destroyed during a system run.

Additionally, the corresponding elements from the class diagram (classes, attributes, associations, association ends) are also included and connected to their instances. These elements exist throughout a system run. An object, for instance, is connected to its class. Each attribute link is connected to an attribute and each object is connected to an attribute link for each attribute its class or one of its superclasses contains. Thus, attributes are inherited from superclasses to subclasses.

In addition to the common snapshot information, there are (object) states in a system state connected to objects. An object of a class, for which a statechart is given, can be connected to a state that has a name referring to a state from the statechart.

When an object is created during a system run there has to be a way to determine the initial state of the newly created object. This is accomplished by connecting the initial state (according to the statechart) to the class of the object.

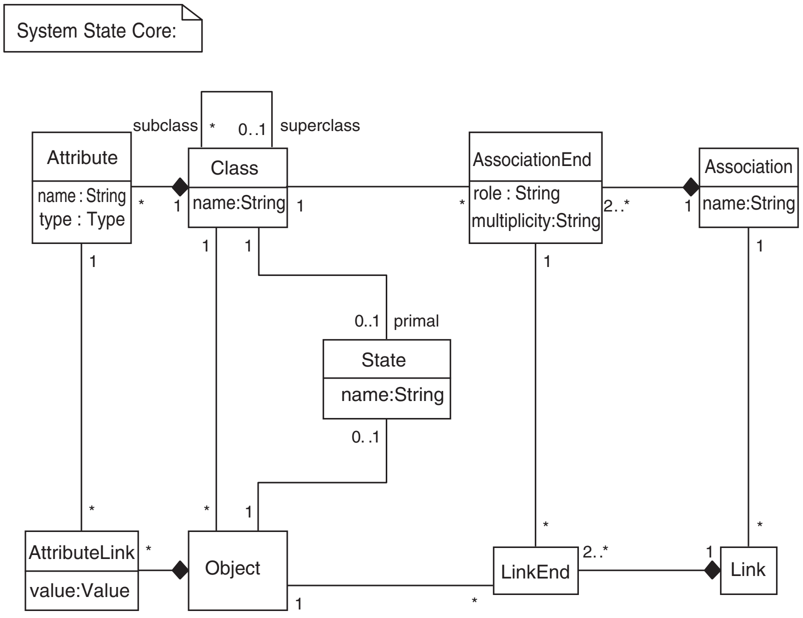


Fig. 11. Abstract syntax of system states—the basics.

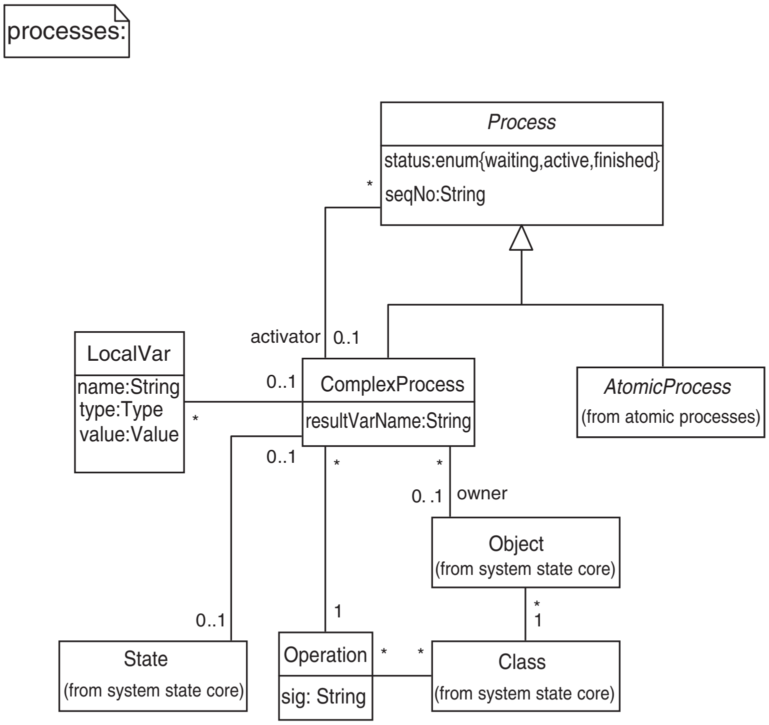


Fig. 12. Kinds of processes.

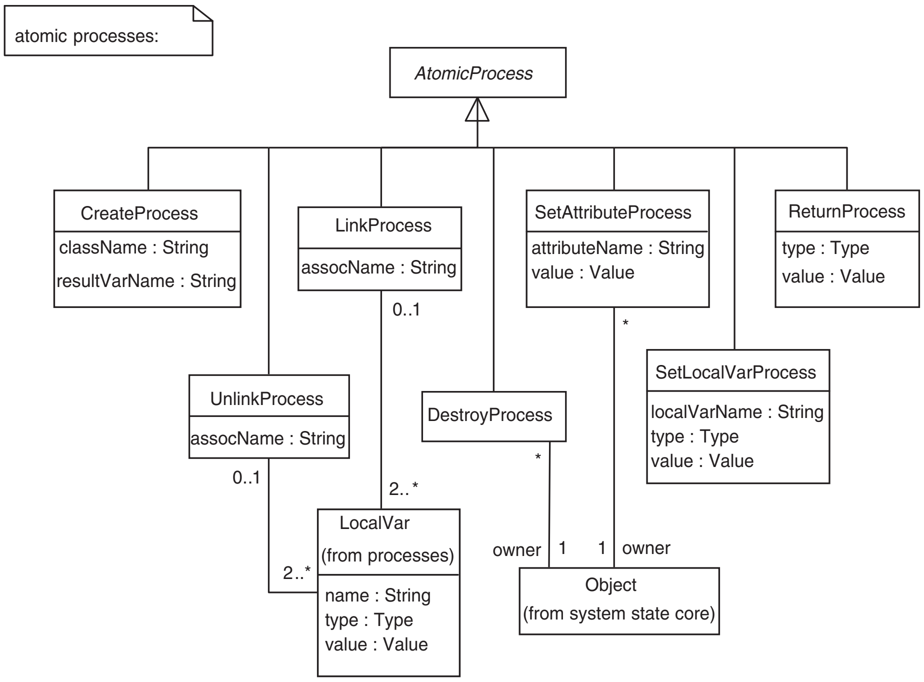


Fig. 13. Atomic processes.

5.1. OCL in graph transformation

As explained in the previous section a system state represents a current snapshot state of a system modeled by a given UML model. This UML model contains the *object model* $\mathscr{M}$ defined in [2]. The object model defines all the available types as well as valid expressions, thus we use $\Sigma_{\mathscr{M}}$ (being the signature of $\mathscr{M}$) as the set of data value nodes in the following. For this reason vertices in rules are attributed with OCL values like 42, *Sequence*$(1,2,3)$ or $o2$ (being an object). A system state may only contain constant values, thus the OCL expressions in the rules have to be evaluated. This is achieved by employing $I(\Sigma_{\mathscr{M}})$ as defined in [2], which maps each type to a carrier set and each operation symbol to a function. The function for evaluating these expressions is not straightforward, as it depends not only on the variable assignment but also on a current system state, which in our case is represented by the host graph. The evaluation of an OCL term e of type t is defined in [2] by a function $I[[e]] : Env \to I(t)$, where Env is a set of environments (σ, β) consisting of a state σ of the system and a variable assignment β. We use this definition, where β is determined by the match and σ is derived from the current system state. This derivation is straightforward since the system state contains all the information included in a system state as defined in [2] extended by current object states. Thus an oclInState expression can be evaluated in our context as well.

The possibility of evaluating OCL expressions in the context of the current host graph yields a comfortable means of formulating further positive application conditions for a rule. An *OCL application condition* (AC) is a Boolean OCL term, specifying a constraint on a match. A match m satisfies an AC e if $I[[e]]$ evaluates to true in the context of the current host graph and the variable binding determined by m. So a rule with an AC may only be applied if a match is found that satisfies the AC.

5.2. Processes

A very important concept of system states is that of a *process*. Roughly speaking, a process represents an operation in execution. As shown in Fig. 12 there are two different kinds of processes: *complex processes* representing user-defined operations in execution and *atomic processes* representing predefined operations in execution.

A user-defined operation is an operation declared in a class diagram and specified by an interaction diagram or a use case, likewise specified by an interaction diagram. In the first case, a corresponding process is executed on an owner object, which is an object of the class the operation is defined for. A complex process can activate other processes, both complex and atomic, and it can have local variables, i.e. variables only visible in the scope of the process. All processes have a sequence number and a status. The sequence number (seqNo) refers to the message in an interaction diagram that corresponds to the process. The status can be waiting, active or finished. A waiting process represents a called operation that has not been started yet. A finished process often is a precondition for calling another operation.

A process can also be connected to a state. This is necessary to determine the state of the owner object once the process is finished. Upon termination of that process, the object will be connected to that state.

Fig. 12 also shows that operations are not only connected to one class but possibly to many. An operation is connected to the class for which it is declared in the class diagram and also to all its subclasses that inherit the operation. An operation is not connected to the subclasses that override the operation by having an operation with the same signature. These connections are needed to ensure that the correct operation is executed on an object that inherits or overrides an operation.

Fig. 13 shows the atomic processes corresponding to the predefined messages mentioned earlier. They are called atomic because they do not activate other processes. We need seven different kinds of atomic processes that handle the low-level modification of the system state:

CreateProcess. A create process creates a new object of the class given by className. The new object is returned to the activator process by setting its local variable given by resultVarName.

DestroyProcess. A destroy process removes its owner object from the system state.

LinkProcess. A link process establishes a link between objects. The association to be used is given in assocName and the objects to be linked are given by a set of local variables. Each variable has an association role as name and an object of appropriate type as value.

UnlinkProcess. An unlink process removes a link between objects. The local variables are analogous to the variables of a link process.

SetAttributeProcess. This kind of process sets the attribute given by attributeName of its owner object to a given value.

SetLocalVarProcess. This kind of process sets the local variable given by attributeName of its activator process to a given value. If the variable does not exist yet it is created.

ReturnProcess. A return process finishes its activator process by returning a value to it.

In the following section, we explain how an integrated UML specification is translated into a graph transformation system, i.e., how the initial system state and the rules changing the system state are constructed.

6. Translation into a graph transformation system

In this section we present an informal overview of the translation of the given UML model into a graph transformation system. In general, the graph transformation system corresponding to the given UML model is built in the following way:

- construct the initial system state using information of the use case, class, object, and statechart diagrams;
- create a rule for every use case;
- generate a transformation unit for every method specified in an interaction diagram;
- add the predefined rules and transformation units to the graph transformation system.

In the next section we will describe how the initial system state is constructed. The second section deals with the generation of rules resp. transformation units that depend on the model. The set of predefined rules, which do not depend on the model, are introduced in the third section. Finally a brief outline of the fundamental design of the prototypic implementation is presented.

6.1. Initial system state

The initial system state contains structural information about the system at the time the program starts. The construction of the initial system state considers the use case-, class-, object-, and statechart diagram supplied by the modeler.

For each use case there is an operation in the system state. It also contains all classes, operations, attributes, associations and association links from the class diagram. The initial state in the statechart diagram for a class is connected to this class. Operations declared in a class are connected to this class and the subclasses that do not override them.

The objects from the object diagram are added together with their links and attribute links. The objects are attached to states according to the notes from the object diagram or (in case there is no note at an object with an associated statechart diagram) to the initial state of the corresponding statechart diagram.

Fig. 14 shows a part of the initial system state of the drive-through example as object diagram instantiating the system state metamodel. Typically, system states in this notation are very large and difficult to handle for human beings. Therefore, a tool like the one presented in Section 6.5 would depict a system state in a more comprehensible way, e.g. by hiding class, attribute and operation vertices as it is usually done with object diagrams. The graph transformation rules, however, work on this complex structure.

Due to the complexity of the graph, Fig. 14 shows only an excerpt of it. In the upper left, you can see an Operation vertex that represents the use case startDriveThrough. The objects d, c1 and c3 represent a drive-through resp. two clients (the other two clients are not shown). The dashed arrows just indicate the relations of the objects, abstracting from the links and link ends that actually constitute these relations.

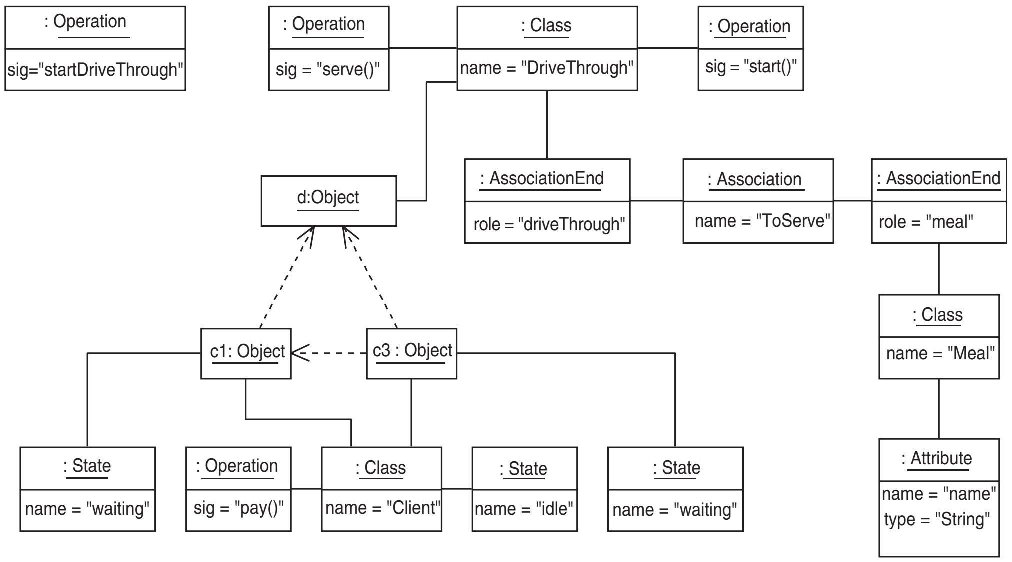


Fig. 14. Start system state for the drive-through example.

During a system run, the system state is modified by graph transformation rules. Basically we need two kinds of rules: rules that depend on the given model and rules that do not, i.e., predefined rules. The following two subsections discuss these rules and how to construct them.

6.2. Rules depending on the model

The initial system state does not contain any processes, i.e., there is no operation that is called and waiting to be executed. This is what the use cases are needed for: for every use case we construct a rule that adds a ComplexProcess vertex with local variables for holding the arguments where necessary.

Fig. 15 shows the rule for the use case startDriveThrough. The rule creates a new ComplexProcess connected to the Operation with the name startDriveThrough. The status is set to waiting and the sequence number is set to 0 (because this process is not created by a message from an interaction diagram).

With this kind of rule, we are now able to add processes to the system state in order to actually start a system run. Next we need rules that handle these processes, i.e., change the system state according to the semantics specified in the interaction diagrams. For every operation specified by an interaction diagram, we construct a set of rules. This is done for every operation no matter whether it belongs to a class or to a use case.

A user-defined operation calls several other operations. An interaction diagram specifies which other operations are called and in which order this has to be done. So an interaction diagram contains messages that are sent between classifier roles in a specific order. Each message represents the call of either a user-defined operation (of a class) or it represents the call of a predefined operation (like setting an attribute value). Every sent message corresponds to the creation of a new process vertex.

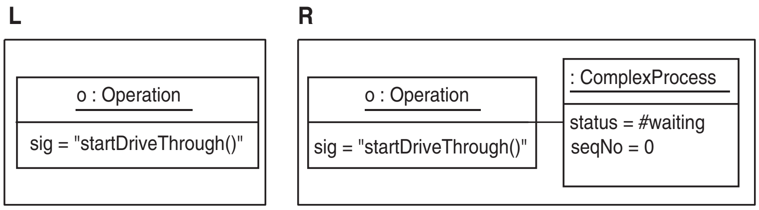


Fig. 15. Rule for creating a use case process.

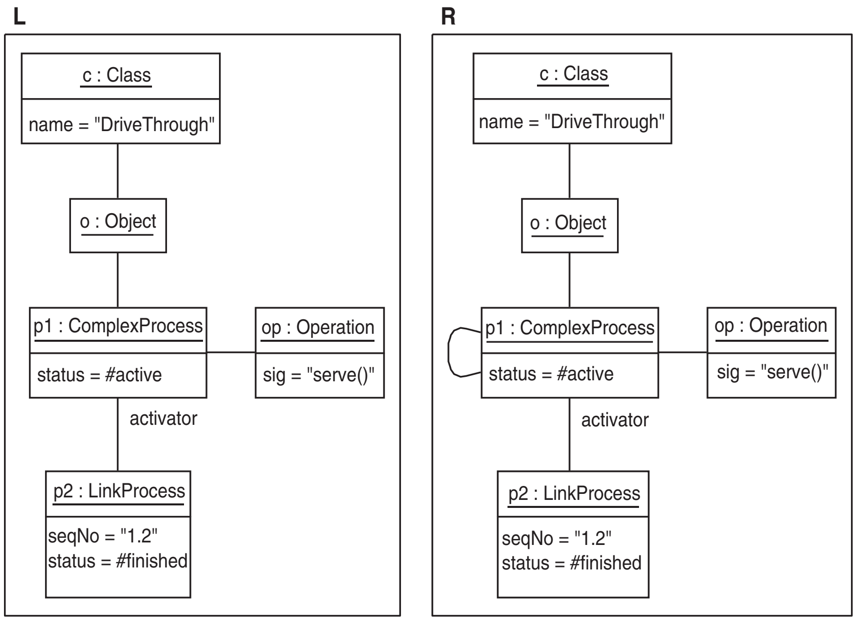


Fig. 16. Sending a message: step 1.

As an example we discuss the transformation unit that is necessary to handle the sending of the message 1.3 of the interaction diagram for DriveThrough::serve() as depicted in Fig. 9.

The message 1.3 is sent during the execution of the operation 1:serve of class DriveThrough. Therefore the new process vertex should only be created if there actually is a process (p1) of this operation (op) running on an object (o) of the desired class (c). This process represents the activator message of the message that corresponds to the process vertex to be created. This activator process vertex is marked with a loop for further rule application. Since in this case the regarded message is not the first one sent by the activator message, the status of the activator process vertex has to be #active. Furthermore the execution of the operation 1.2 should have been finished, thus the status of the corresponding process vertex (p2)must be finished. The first rule r1 depicted in Fig. 16 does exactly that.

The second rule r2 creates a waiting process. This can be regarded as actually sending the message (see Fig. 17). The considered message in our example is sent to an object

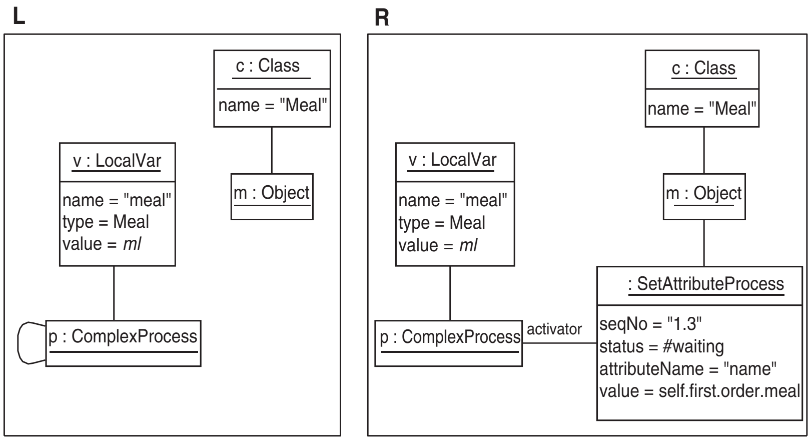


Fig. 17. Sending a message: step 2.

stored in a local variable meal as indicated by the stereotype ≪ local ≫ and the role name meal. In order to attach the new waiting process vertex to the correct object vertex, the left-hand side of the rule demands the presence of the LocalVar vertex attached to the activator process vertex. This LocalVar vertex has the value Meal which refers to the Object vertex connected to the class vertex with the name Meal. The new process vertex with status #waiting will be connected to that object vertex and its activator process vertex. Its other attributes are set according to the collaboration diagram. In particular, the value is set to an OCL expression that is evaluated while applying the rule. In addition, the rule removes the loop from the activator process. The two rules have to be applied one after the other, thus the control condition of the transformation unit is r1;r2, meaning that first the rule r1 and after it rule r2 has to be applied.

In general, there are a lot of circumstances to consider when constructing such a transformation unit for sending a message. After having shown the concrete example, we will only outline some details not covered by the example.

- When a complex process is created instead of an atomic one, it has to be connected to the correct Operation vertex. This is the reason why two rules are needed in general, otherwise there would be two Operation vertices in one rule that possibly represent the same operation. Because we use injective matching, this would be impossible.
- Asynchronous predecessors are represented by processes with no specified status in the rule, i.e., an asynchronous predecessor does not have to be finished.
- Parallel messages are sent by creating corresponding processes at the same time in one rule.
- If the message to be sent is the message that starts the execution of a waiting process and this execution is allowed only in certain states according to a statechart diagram, then the rule must only be applicable with a match containing a receiver object in the correct state. The rule then disconnects the state from the object, which is then "between" two states, and connects the next state to the new process. The object obtains its new state by another rule when the process has finished.

- If a message is sent via an association role without stereotype, a process is created and connected to an arbitrary object (of the specified class) that is linked to the sending object (by a link of the specified association). Thus, the rule includes links, associations, link ends, etc. In addition, the rule creates a new local variable that stores the chosen object, in case other messages are sent via the same association role.

6.3. Predefined rules

Predefined messages do not call a user-defined operation but rather a predefined operation. There are messages for creating an object of a specific class, destroying an object, connecting objects with a link of a given association, unlinking objects, setting an attribute value, setting a local variable value, and returning a result. Corresponding to these messages there are atomic processes that are not associated with an operation but instead with other information needed for the task. These atomic processes have already been shown in Fig. 13. The rules that handle these processes are also predefined, i.e., they are independent from the user model.

There is a predefined rule or even transformation unit for each kind of atomic process. In addition to these, there are rules for collecting garbage. These rules remove finished processes that are no longer needed as preconditions for other processes, and local variables that are no longer attached to a process vertex. These rules are applicable whenever such garbage exists in the system state.

In the following, we will present the transformation unit for handling a link process in more detail.

A LinkProcess has several local variables, each of them indicating the object that is supposed to play a certain role in the link that shall be created. The following transformation unit manages to create and connect a Link vertex and LinkEnd vertices that connect the objects as requested.

link

rules:	*createLink*	(Fig. 18)
	createLinkEnds	(Fig. 19)
	finishLinking	(Fig. 20)
cond:	*createLink*; *createLinkEnds*!; *finishLinking*	

To execute a LinkProcess, the rule *createLink* is first applied to create a Link vertex with a flag. This new vertex is connected to the association indicated by assocName. This rule also changes the status of the process to #active. Then, the rule *createLinkEnds* is applied as long as possible to create LinkEnd vertices for all objects that are to be linked and to connect these vertices. The NAC ensures that only one link end is created per association end. This rule is supposed to attach new LinkEnd vertices only to the Link vertex that was created earlier in the application of this transformation unit, which is ensured by the flag. The flag is removed and the process is finished by the rule *finishLinking* (see Figs. 18–20).

In general the use of transformation units is employed in order to encapsulate functionality into transactions. This is due to the fact that certain rules cannot change the system state in one step. Since the intermediate graphs are of no interest (especially since the graph transformation background of the actual system state run is hidden from the modeler) they are thus neither visible nor accessible to the modeler. The information from the model is not reflected in the structure of the control conditions. Therefore information from the model (e.g. the order of message calls) is still needed in the rules (e.g. sequence number).

6.4. Inheritance

In a system state an object has attribute values that are stored in AttributeLink vertices for all attributes that are declared in its class as well as in classes it inherits. The creation of the initial system ensures this, and the transformation unit that handles the creation of a new object also respects this fact. Therefore, a class inherits the attributes of its superclasses. Associations are implicitly inherited by the construction of the transformation unit that realizes a Link operation (for more details cf. [7]). An important property of

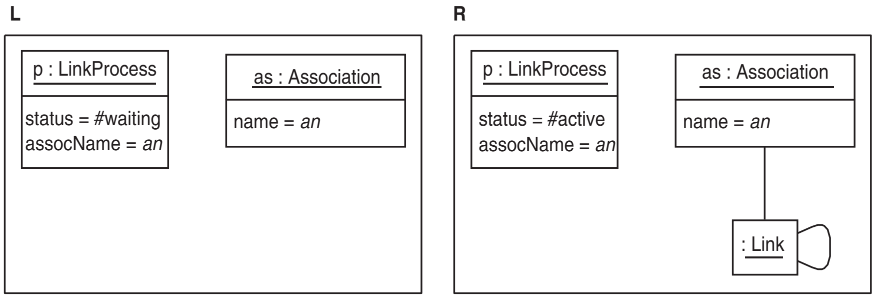


Fig. 18. Rule *createLink*.

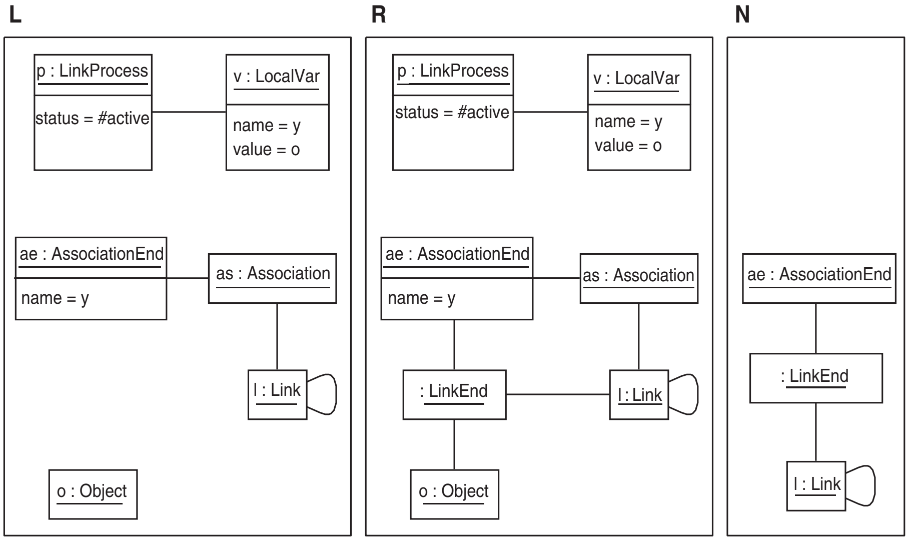


Fig. 19. Rule *createLinkEnds*.

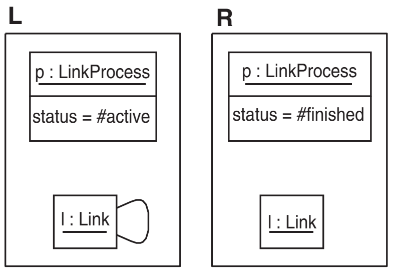


Fig. 20. Rule *finishLinking*.

object-oriented systems that is supported by our approach is *subtype polymorphism*. Here an operation may be performed differently in different classes all of which have a common superclass with said operation. In the context of our approach this happens whenever a message is sent to a classifier role associated with a class that has subclasses which override the called operation. In this case either the operation of the superclass or one of the overriding operations is called, depending on the class of the object that actually receives the message. We only allow to override an operation with another operation that has exactly the same signature (called *invariant overriding* (cf. e.g. [39]) in order to avoid typing problems. Such an overridden method has to be specified in its own interaction diagram.

This is possible, since an Operation vertex is not only connected to the one class that declares it, but also to all of its subclasses that do not override it.

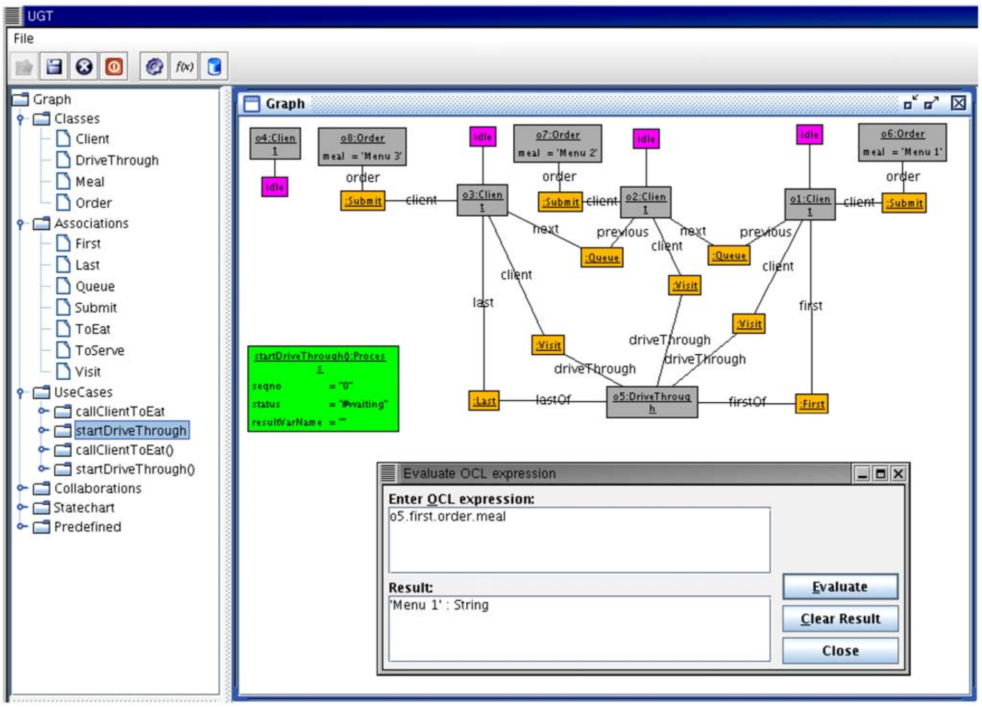
6.5. Implementation

Currently a prototype for the concepts discussed in this paper is being implemented. The goal of this prototype is to visualize the evolution of the system state. When provided with a model, the prototype automatically generates the graph transformation rules and the initial system state graph. A graphical user interface then permits the user to view the evolution of the system state step by step and to examine the current state by querying it using OCL.

For this reason the prototype must be able to perform graph transformations as well as to evaluate OCL expressions. Instead of implementing a new tool for these purposes, we chose to combine two well established tools. The graph transformation part is done by AGG [40] and the evaluation of OCL expressions is performed by the USE tool [41]. An obvious choice for the tool implementation would have been the Fujaba tool [22], which already couples graph transformation and UML. But since the UML part has to be handled by the USE tool anyway (in order to evaluate OCL expressions), any graph transformation engine capable of applying the rules generated by our approach suffices. Since the developers of the prototype are very familiar with AGG and its API, it has been the preferred choice as underlying graph transformation engine.

The prototype (hence UGT—*U*ML to *G*raph *T*ransformation) reads a USE specification of a UML model that is compatible with our approach. It then generates the set of graph transformation rules according to the ideas presented in this paper. Additionally, the initial host graph is constructed from the object diagram the modeler provides. The GUI of UGT then displays this initial graph and the use case names as specified in the model. The user may now select a use case to be executed. In this case, the rule that starts the execution of this use case is applied to the host graph. Now the user may click the step button and thus derive a next step in the system state evolution. Internally UGT calculates the next step by randomly choosing one of the rules that are applicable and letting AGG apply it. This may be done until no further rule is applicable. If this is the case, the use case is completely finished. As well as letting the system decide upon the next step, the user may control the flow of execution. Imagine a state with two processes in status waiting or active that can both be executed in the next step. In this case the user may decide what process is executed next by simply double-clicking it. If the process is specified to expect parameters, the user is prompted with an input field and forced to provide the necessary parameters. Fig. 21 shows a screenshot of UGT in action. A system state is displayed with three clients visiting a drive-through, each of them with a submitted order. The user chose the use case startDriveThrough to be started, which resulted in the corresponding process being added to the system state.

By executing the system state step by step, the user can gain insight into the modeled system. Furthermore, UGT allows the evaluation of OCL expressions at any step in the system state. The OCL evaluation window displaying an OCL query and its evaluated result can be seen in the lower part of Fig. 21. Due to this feature, it is possible to check whether invariants hold during the execution of operations or even complete use cases. Note that UGT completely hides the graph transformation basis of this approach from the user. They do not need to know about the rule generation, or the fact that graph transformation is used to derive the next step of the system state.



startDriveThrough0:Process
seqno = "0"
status = "#waiting"
resultVarName = ""

Enter OCL expression:
o5.first.order.meal

Result:
'Menu 1' : String

Fig. 21. Screenshot of UGT.

7. Conclusion and future work

We have presented a conceptual approach for defining a semantics for UML based on the translation of a given UML model into a graph transformation system. To demonstrate our approach an example model comprising several UML diagrams has been introduced. Next we have described our idea of a system state by means of a metamodel followed by a discussion of the translation of a given model into model-depending and predefined graph transformation rules by example. Finally the basic concepts of the prototypic software implementing this approach have been addressed. The prototype translates a given UML model into a graph transformation system and allows to monitor the evolution of the system state step by step.

The next goal is to complete the prototype implementation and to further enhance its GUI. As the approach and the tool are suitable for early stages of the software development process, it might become impractical when using large and very detailed models. In this case the aforementioned GUI should allow the user to choose different views on the system run, like e.g. hiding objects and their details that are of no interest in a certain situation.

An interesting topic would be the integration of further diagram types like activity diagrams into our approach. We will also investigate whether and how the diagrams already covered can be extended with yet missing UML features. These include composite states and concurrent ones in statechart diagrams and ≪include≫ and ≪extend≫ relationships between use cases.

It may also be worth investigating whether a set of elementary templates can be provided for the rules that depend on the model. Currently every rule has to be generated from scratch for every model. Rule templates would provide a better maintainability in the sense that central concepts could be changed in one place instead of different rules.

Case studies will provide feedback on the practicability of the approach and tool. In particular, more insight is needed into the process of asserting properties of UML models on the basis of our approach, for instance, based on transformation invariants. In this way our approach will automatically benefit from future results in the field of graph transformation.

References

[1] OMG, OMG Unified Modeling Language Specification, Version 1.5, March 2003, Object Management Group, Inc., Framingham, MA, <http://www.omg.org>, 2003.

[2] M. Richters, A precise approach to validating UML models and OCL constraints, Ph.D. Thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[3] Boldsoft, Rational Software Corporation, and IONA, Response to the UML 2.0 OCL RfP (ad/2000-09-03), January 2003. <http://www.klasse.nl/ocl/ocl-subm.html>.

[4] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation, Foundations, vol. 1, World Scientific, Singapore, 1997.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, Applications, Languages and Tools, vol. 2, World Scientific, Singapore, 1999.

[6] H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, Concurrency, Parallelism, and Distribution, vol. 3, World Scientific, Singapore, 1999.

[7] P. Ziemann, An integrated operational semantics for a UML core based on graph transformation, Ph.D. Thesis, University of Bremen, 2005.

[8] S.-K. Kim, D.A. Carrington, An integrated framework with UML and Object-Z for developing a precise specification: the light control case study, in: 7th Asia-Pacific Software Engineering Conference (APSEC 2000), 5–8 December 2000, Singapore. IEEE Computer Society, 2000, pp. 240–248.

[9] T. Clark, A. Evans, S. Kent, The metamodelling language calculus: foundation semantics for UML, in: H. Hussmann (Ed.), Fundamental Approaches to Software Engineering, Fourth International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2–6, 2001, Proceedings, Lecture Notes in Computer Science, vol. 2029, Springer, Berlin, 2001, pp. 17–31.

[10] A. Evans, S. Kent, Core meta-modelling semantics of UML: the pUML approach, in: R. France, B. Rumpe (Eds.), UML'99—The Unified Modeling Language, Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28–30, 1999, Proceedings, Lecture Notes in Computer Science, vol. 1723, Springer, Berlin, 1999, pp. 140–155.

[11] R.B. France, A. Evans, K. Lano, B. Rumpe, The UML as a formal modeling notation, Computer Standards and Interfaces 19 (7) (1998) 325–334.

[12] H. Störrle, Semantics and verification of data flow in UML 2.0 activities, in: M. Minas (Ed.), Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM 2004), Electronic Notes in Theoretical Computer Science, vol. 127(4), Elsevier, Amsterdam, 2005, pp. 35–52.

[13] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, V. Thurner, Towards a formalization of the Unified Modeling Language, in: M. Aksit, S. Matsuoka (Eds.), ECOOP'97—Object-Oriented Programming, 11th European Conference, Lecture Notes in Computer Science, vol. 1241, Springer, Berlin, 1997, pp. 344–366.

[14] R. Eshuis, R. Wieringa, A real-time execution semantics for UML activity diagrams, in: H. Hussmann (Ed.), Fundamental Approaches to Software Engineering, Fourth International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2–6, 2001, Proceedings, Lecture Notes in Computer Science, vol. 2029, Springer, Berlin, 2001, pp. 76–90.

[15] R. Wieringa, Formalizing the UML in a systems engineering approach, in: H. Kilov, B. Rumpe (Eds.), Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications), Technische Universität München, TUM-I9813, 1998, pp. 254–266.

[16] L. Starr, Executable Uml: How to Build Class Models, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001 (Foreword by Stephen J. Mellor).

[17] S. Kuske, M. Gogolla, R. Kollmann, H.-J. Kreowski, An integrated semantics for UML class, object, and state diagrams based on graph transformation, in: M. Butler, K. Sere (Eds.), Third International Conference on Integrated Formal Methods (IFM'02), Lecture Notes in Computer Science, vol. 2335, Springer, Berlin, 2002, pp. 11–28.

[18] M. Gogolla, P. Ziemann, S. Kuske, Towards an integrated graph based semantics for UML, in: Graph Transformation and Visual Modeling Techniques (GT-VMT 2002), ENTCS, vol. 72, 2003.

[19] R. Heckel, S. Sauer, Strengthening uml collaboration diagrams by state transformations, in: H. Hussmann (Ed.), Fundamental Approaches to Software Engineering, Fourth International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2–6, 2001, Proceedings, Lecture Notes in Computer Science, vol. 2029, Springer, Berlin, 2001, pp. 109–123.

[20] S. Kuske, A formal semantics of uml state machines based on structured graph transformation, in: M. Gogolla, C. Kobryn (Eds.), UML 2001—The Unified Modeling Language, Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, vol. 2185, 2001, pp. 241–256.

[21] D. Varró, A formal semantics of UML statecharts by model transition systems, in: A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 2002, Proceedings, Lecture Notes in Computer Science, vol. 2505, Springer, Berlin, 2002, pp. 378–392.

[22] T. Fischer, J. Niere, L. Torunski, A. Zündorf, Story diagrams: a new graph transformation language based on UML and Java, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Proceedings of the Theory and Application to Graph Transformations (TAGT'98), Paderborn, November, 1998, Lecture Notes in Computer Science, vol. 1764, Springer, Berlin, 1998.

[23] G. Engels, R. Heckel, J.M. Küster, L. Groenewegen, Consistency-preserving model evolution through transformations, in: J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.), UML 2002—The Unified Modeling Language, Model Engineering, Languages, Concepts, and Tools, Fifth International Conference, Dresden,

Germany, September/October 2002, Proceedings, Lecture Notes in Computer Science, vol. 2460, Springer, Berlin, 2002, pp. 212–226.
[24] A. Schürr, A.J. Winter, UML Packages for PROgrammed Graph REwriting Systems, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), TAGT, Lecture Notes in Computer Science, vol. 1764, Springer, Berlin, 1998, pp. 396–409.
[25] A. Tsiolakis, H. Ehrig, Consistency analysis of UML class and sequence diagrams using attributed graph grammars, in: H. Ehrig, G. Taentzer (Eds.), Proceedings of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Berlin, March 2000, Technical Report no. 2000/2, Technical University of Berlin.
[26] E. Börger, A. Cavarra, E. Riccobene, Modeling the dynamics of UML state machines, in: Y. Gurevich, P.W. Kutter, M. Odersky, L. Thiele (Eds.), Abstract State Machines, Lecture Notes in Computer Science, vol. 1912, Springer, Berlin, 2000, pp. 223–241.
[27] A. Cavarra, E. Riccobene, P. Scandurra, Integrating UML static and dynamic views and formalizing the interaction mechanism of UML state machines, in: E. Börger, A. Gargantini, E. Riccobene (Eds.), Abstract State Machines, Lecture Notes in Computer Science, vol. 2589, Springer, Berlin, 2003, pp. 229–243.
[28] E. Börger, A. Cavarra, E. Riccobene, An ASM semantics for UML activity diagrams, in: T. Rus (Ed.), Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST'2000), Lecture Notes in Computer Science, vol. 1816, Springer, Berlin, 2000, pp. 293–308.
[29] S. Flake, W. Mueller, An ASM definition of the dynamic OCL 2.0 semantics, in: T. Baar, A. Strohmeier, A. Moreira, S.J. Mellor (Eds.), Proceedings of the International Conference on Unified Modeling Language (UML'2004), Lecture Notes in Computer Science, vol. 3273, Springer, Berlin, 2004, pp. 226–240.
[30] A. Cavarra, J.K. Filipe, Formalizing liveness-enriched sequence diagrams using ASMs, in: W. Zimmermann, B. Thalheim (Eds.), Abstract State Machines, Lecture Notes in Computer Science, vol. 3052, Springer, Berlin, 2004, pp. 62–77.
[31] W. Shen, K.J. Compton, J. Huggins, A UML validation toolset based on abstract state machines, in: ASE, IEEE Computer Society, Silver Spring, MD, 2001, pp. 315–318.
[32] W. Shen, K.J. Compton, J. Huggins, A method of implementing UML virtual machines with some constraints based on abstract state machines, in: APSEC, IEEE Computer Society, Silver Spring, MD, 2003, pp. 224–232.
[33] S. Sendall, A. Strohmeier, From use cases to system operation specifications, in: A. Evans, S. Kent, B. Selic (Eds.), UML 2000—The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2000, Proceedings, Lecture Notes in Computer Science, vol. 1939, Springer, Berlin, 2000, pp. 1–15.
[34] R. Heckel, J.M. Küster, G. Taentzer, Confluence of typed attributed graph transformation systems, in: A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7–12, 2002, Proceedings, Lecture Notes in Computer Science, vol. 2505, Springer, Berlin, 2002, pp. 161–176.
[35] H.-J. Kreowski, S. Kuske, Graph transformation units with interleaving semantics, Formal Aspects of Computing 11 (6) (1999) 690–723.
[36] S. Kuske, Transformation units—a structuring principle for graph transformation systems, Ph.D. Thesis, University of Bremen, 2000.
[37] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, Reading, MA, 1998.
[38] B. Cordes, K. Hölscher, H.-J. Kreowski, UML interaction diagrams: correct translation of sequence diagrams into collaboration diagrams, in: M. Nagl, J. Pfaltz, B. Böhlen (Eds.), AGTIVE'03 Proceedings, Lecture Notes in Computer Science, vol. 3062, Springer, 2004, pp. 275–291.
[39] M. Abadi, L. Cardelli, A Theory of Objects, Springer, Berlin, 1998.
[40] The Attributed Graph Grammar System AGG, last revision 2005. <http://tfs.cs.tu-berlin.de/agg>.
[41] A UML-based Specification Environment, last revision 2005. <http://www.db.informatik.uni-bremen.de/projects/USE>.