An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation*

Sabine Kuske, Martin Gogolla, Ralf Kollmann, Hans-Jörg Kreowski

University of Bremen, Department of Computer Science
P.O.Box 330440, D-28334 Bremen, Germany
{kuske,gogolla,kollmann,kreo}@informatik.uni-bremen.de

Abstract. This paper studies the semantics of a central part of the Unified Modeling Language UML. It discusses UML class, object and state diagrams and presents a new integrated semantics for both on the basis of graph transformation. Graph transformation is a formal technique having some common ideas with the UML. Graph transformation rules are associated with the operations in class diagrams and with the transitions in state diagrams. The resulting graph transformations are combined into a one system in order to obtain a single coherent semantic description.

1 Introduction

In recent years, the Unified Modeling Language UML [BRJ98,RJB98,OMG01] has been widely accepted as the standard language for modeling and documenting software systems. The UML offers a number of diagram forms used to describe particular aspects of software artifacts. These diagram forms can be divided depending on whether they are intended to describe structural or behavioral aspects. From a fundamental point of view, class and state diagrams are the basic means for system description, because class diagrams determine the fundamental object structures and state diagrams can be employed for describing the fundamental object behavior.

Graph transformation is a well-developed field [Roz97,EEKR99,EKMR99] and has many application domains, among them questions related to graphical modeling languages like UML [SW00,EHHS00,FNTZ98,Gog00,PS00,TE00], [EHK01,Kus01]. The basic idea of graph transformation is to represent system states as graphs and system evolution steps as applications of single graph transformation rules. We feel that this basic idea has many similarities with basic UML concepts. System states in UML can be regarded as object diagrams which

* This work was partially supported by the project Abstract Implementation of and Documentation with UML (UML-AID) funded by the *Deutsche Forschungsgemeinschaft* and the ESPRIT Working Group Applications of Graph Transformation (APPLIGRAPH).

belong to certain class diagrams. Object diagrams are essentially graphs. System evolution in UML may be described in various ways. One basic behavior description technique is the UML state diagram where each transition may be regarded as an atomic system evolution step transforming a system state, i.e. a graph, into another system state. Therefore, concepts of UML and graph transformation are closely related. Another advantage we see in regarding graphs as a semantic description technique is that graphs can be understood as visual entities like all diagrams in the UML. Explaining UML by graphs induces only a small gap between the language to be defined, namely the UML, and the language used as the target language, namely graphs.

The structure of the rest of the paper is as follows. Section 2 discusses the features of UML class and state diagrams we use in this paper. Section 3.1 explains how class diagrams can be translated to graph transformation and Sect. 3.2 does the same for state diagrams, in particular Sect. 3.2 shows how state transitions are expressed. Section 4 discusses how to integrate the resulting graph transformations from the class and state diagrams into a single transformation system. All concepts are explained by a single running example. Section 5 shortly touches related work. The paper closes in Sect. 6 with some final remarks.

2 Class and state diagrams

Class and state diagrams are fundamental diagrams of the UML. In the following we shortly illustrate both diagram types. For further details, the reader is referred to e.g. [BRJ98,RJB98,OMG01].

Class diagrams are used to represent the static structure of object-oriented systems. They consist of classes and relationships where the latter are divided into associations, generalizations, and dependencies. Special kinds of associations are compositions and aggregations. A *class* consists of a name, a set of attributes and a set of operations possibly with parameters. Every class c specifies a set of objects called the *instances of* c. An *association end* is a language element of class diagrams which relates associations with classes and contains some information such as the *role* a class plays in the corresponding association or its *multiplicity.* A *class diagram* is a graph where the nodes are classes, and the edges are associations, generalizations, or dependencies. We here concentrate on binary associations. Some of the classes may be associations as well. These classes are called association classes.

Fig. 1 shows an example of a class diagram consisting of classes, binary associations, and an association class. It models a client-server system containing a class *DriveThrough* and a class *Client.* A drive-through can be visited by an ordered set of clients (client queue). Every client of a drive-through has a running number which indicates the place the client has in the client queue of the drive-through. Furthermore, there is a class *Order* which represents possible orders a client can give to a drive-through.

Object diagrams mainly differ from class diagrams in the sense that they contain objects instead of classes. Object diagrams are useful to represent the

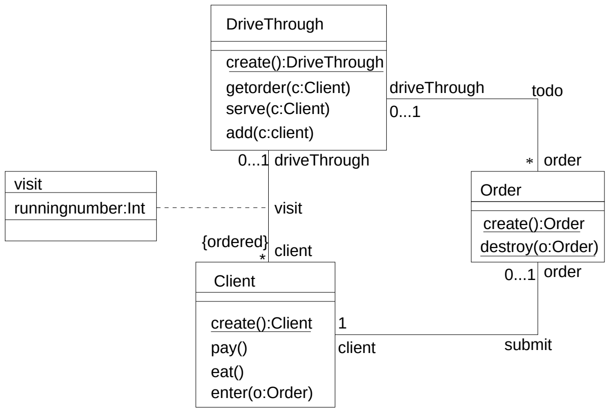

Fig. 1. A class diagram

state of a system in a special moment. The nodes of an object diagram are objects and its edges are links. Analogously to class diagrams some objects may be links as well. An example of an object diagram is shown in Fig. 2. It contains five objects: one drive-through *McD*, three clients *Ada*, *Bob*, and *Cher*, as well as one order *Shake*. The order is linked to the drive-through and to client *Ada*. Intuitively, this means that client *Ada* has submitted order *Shake* at the drive-through which in turn has to do the serving of this order. Moreover, the two *visit*-links signify that clients *Ada* and *Bob* are visiting the drive-through. The numbers beneath the links are the running numbers of the visits. Client *Cher* is not related to the drive-through.

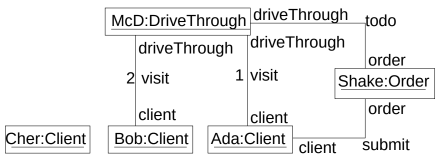

Fig. 2. An object diagram

An object diagram *fits* a class diagram if the objects are instances of the classes, the links can be mapped to the associations, and the multiplicity requirements of the class diagram are satisfied. For example the object diagram of Fig. 2 fits the class diagram in Fig. 1 but drawing an additional link from *Shake* to *Bob* would violate the specified multiplicities.

The dynamic behavior of object-oriented systems can be modeled with UML state diagrams which, in general, can be associated with classes in order to describe the behavior of the objects of the classes. A *state diagram* consists of a set of states and a set of transitions connecting them. The states are object states and the transitions specify state changes. In the following a simplified kind of UML state diagrams is considered that allows us to illustrate the basic ideas of defining an integrated semantics for UML class and state diagrams in a suitable way.

In this simplified model of state diagrams a *state* is just a name and a *transition* connects two states in a directed way. It is labeled with an event, a guard, and an action. A *guard* is an OCL expression (cf. [WK98,OMG01]). OCL expressions may be logic formulas that evaluate either to *true* or to *false*. Events and actions can be of many types [RJB98]. In the following we restrict ourselves to call events and a simple kind of call actions. A (*call*) *event* is of the form $op(p_1, \ldots, p_n)$ where *op* is a name (of an operation) and $p_1, \ldots, p_n$ are instantiated parameters. If *op* is an operation of a class c $op(p_1, \ldots, p_n)$ is called an *event of* c. A (call) action is of the form $o.e$ where o is an expression which specifies at run-time an object of some class and e is a call event of that class (see Fig. 3). To put it in another way, let us assume that there exists a class diagram CD containing a class c and that there is a state machine associated with c. Let $o.e$ be a call action in a transition of the state diagram associated with c. Then we require that o specifies an object of class c' in the context of class c. Moreover, e should be a call event of the class c' and we say that the action $o.e$ is a call action of the class c.

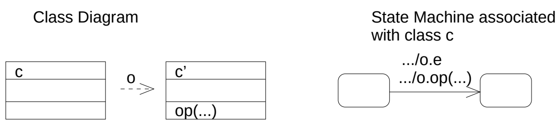

Fig. 3. Illustration of a call action

Examples of state diagrams for the classes *Client* and *DriveThough* of the above class diagram are depicted in Fig. 4. The objects of class *Client* can be in the state *ClientLife*, *HasOrderd*, or *hasPaid* and the transitions specify changes between those states. Analogously, any *DriveThrough* can be in the states *DriveThroughLife* or *ReceivedOrder*.

The firing of transitions is part of the execution semantics of state diagrams which is based on so-called run-to-completion steps. Let STD be a state diagram associated with some class c. Let o be an object of class c which is in state s.[1] We assume that the object o has an associated event queue q, and let e be the next event in q to be dispatched. Then in a run-to-completion step one transition

[1] Note that initially every object is in the target state of the transition which has the initial state as source. This means for our running example that every client is initially in the state *ClientLife* and every drive-through in the state *DriveThroughLife*.

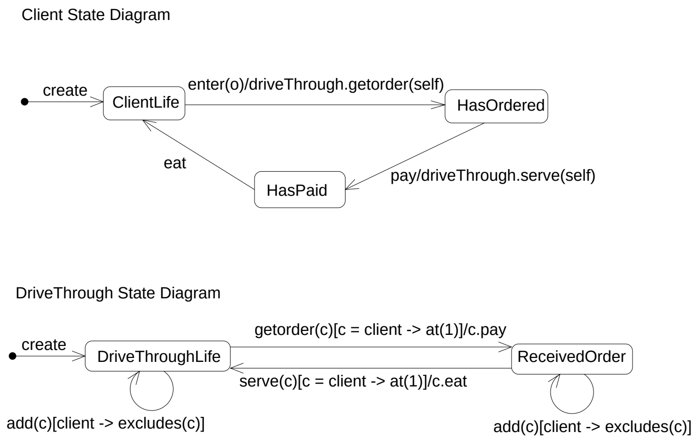

Fig. 4. State diagrams for *Client* and *DriveThrough*

with source s is chosen provided that it is labeled with event e and its guard can be evaluated to *true*. The firing of the transition includes the execution of e and sends the call event of a possible call action of the transition to the object specified in the call action. Afterwards STD is in the target state of the transition.

As an example consider the state diagrams in Fig. 4. Roughly speaking, and considering the events of the transitions the life of a *Client* consists of entering orders, paying, and eating, and the life of a *DriveThough* consists of getting and serving orders. Assume that there exists a *Client* object c being in the state *ClientLife* and that the next event in the event queue of c is $enter(o)$ where o is some order. Then the transition with source *ClientLife* can fire. This means that the call event $getorder(c)$ is sent to the drive-through visited by c. Afterwards the client is in the state *HasOrdered*. Note that according to the UML specification a dispatched event is just deleted from the event queue if no transition can fire. In this case the state diagram remains in its previous state. Note that in the context of class *Client*, the OCL expression *driveThrough* (role name of visit) is of type *DriveThrough* (due to the multiplicity $0 \ldots 1$), and in the context of class *DriveThrough* the OCL expression *client* is of type *Sequence*(*Client*) (according to the multiplicity $*$ and the specification *ordered*).

3 Graph transformation rules for class and state diagrams

Graph transformation originated about thirty years ago as a generalization of the well-known Chomsky grammars to graphs. It is a theoretically well studied area with many potential application domains (see [Roz97,EEKR99,EKMR99] for an overview). In the following we briefly present the basic concepts of graph transformation.

The basic operation of graph transformation consists of the local manipulation of graphs via the application of a rule. A *graph* consists of a set of (labeled) nodes and a set of (labeled) edges connecting them. Examples of graphs are the class diagram and the object diagram presented in the previous section where, roughly speaking, the nodes represent classes, resp. objects, and the edges associations, resp. links. A *graph transformation rule* mainly consists of two graphs, called *left-hand side* and *right-hand side* which have a common part. An example of a rule is depicted in Fig. 5. The left-hand side and the right-hand side are object diagrams. A rule (L, R) is applied to a graph G by choosing an image $g(L)$ of the left-hand side L in the graph G and by replacing $g(L)$ by the right-hand side R such that the common part is maintained. The rule in Fig. 5 can be applied to the object diagram of Fig. 2. It deletes the link between the drive-through and the order.

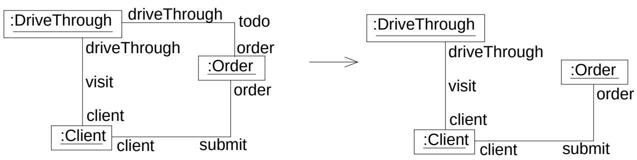

Fig. 5. Rule for *DriveThrough* :: *serve*(c : *Client*)

In the following we are going to illustrate with our running example how graph transformation rules can be associated with the operations of class diagrams and with the transitions of state diagrams. After that we will present in Section 4 how both diagram and rule types can be integrated into a graph transformation system which specifies the integrated semantics of class diagrams with associated state diagrams.

3.1 Associating graph transformation rules with class diagrams

In general, the semantics of class diagrams can be defined as the set of all its object diagrams. Such an object diagram can be interpreted as a state of the system to be modeled, and the execution of operations of the class diagram may modify the state so that another object diagram is obtained. Clearly, this requires that additionally to the semantics of a class diagram, say CD, we specify a semantics for every operation in CD. This semantics is a binary relation on the semantics of CD, i.e. on the set of all object diagrams of CD. For example, we may specify that the operation *serve*(Ada) applied to the object diagram in Fig. 2 deletes the link between *Shake* and *McD*.

Fig. 5 shows the rule which implements the operation *serve*(c), i.e. the application of this rule to the above object diagram deletes the link between the *Order* and the *DriveThrough* object. Analogously we can assign a graph transformation rule to every other operation of our example class diagram (see Fig. 6 where *empty* denotes the empty graph). The rule *DriveThrough*::*create*(): *DriveThrough* creates an arbitrary object of type *DriveThrough*; the rule *Client*::*create*(): *Client* works analogously for clients. The rule *DriveThrough*::*getorder*(c: *Client*) requires that before executing the operation *getorder*(c) the client c is visiting a drive-through and has submitted an order. This is expressed by the left-hand side of the rule. After the execution of *getorder*(c) a *todo*-link must exist between the order and the drive-through. The rules for the remaining operations can be explained in an analogous way.

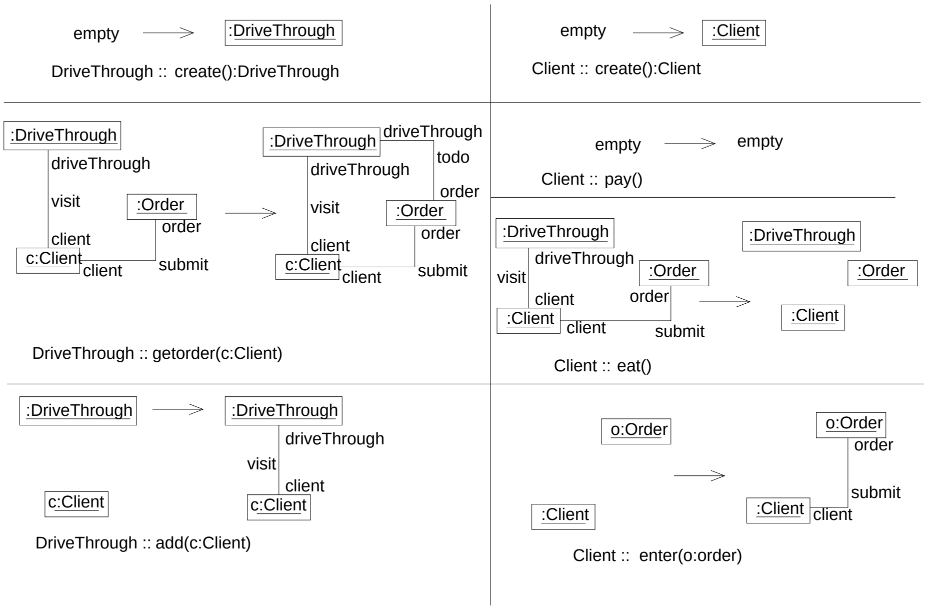

Fig. 6. Further rules for the class diagram of Fig. 1

Summarizing, the presented graph transformation rules provide a means which allows us to specify in a direct and intuitive way how object diagrams (i.e. system states) change after the execution of an operation. Moreover, it is possible to specify pre-conditions for the execution of the operations. In UML those pre- and post-conditions can be specified with OCL expressions.

The above consideration leads to the notion of *extended class diagrams* which consist of a class diagram CD and a mapping *mrule* which assigns a graph transformation rule to every operation of CD.

In general, one cannot assume that an operation can always be associated with a single graph transformation rule which specifies its semantics, because the operation may be too complicated. For those cases, more sophisticated concepts of graph transformation are needed such as, for example, *transformation* *units* that allow to encapsulate sets of graph transformation rules and which provide control mechanisms for the application process of its local rules (cf. [KK99,Kus00]).

3.2 Representing transitions as graph transformation rules

The transitions of a state diagram STD can be also represented by means of graph transformation rules. The main idea associates a graph transformation rule with each transition. Let c be the class the state diagram STD is associated with, and let o be an object of class c. Let $t = (s, e, g, o'.e', s')$ be a transition of STD where s denotes the source state of t, e the event, g the guard, $o'.e'$ the call action, and s' the target state. Then the rule for t should model the dispatching of e in the event queue of o, the change of the state of o from s to s', and the insertion of e' in the event queue of the object specified by o'. For this purpose the rule contains in its left- and right-hand side the object o (i.e. more precisely a variable of type c). It contains also the object or the path specified by o'. The state s is associated with o in the left-hand side of the rule, and the state s' is associated with o in the right-hand side. The event e is connected to o in the left-hand side whereas in the right-hand side e' is connected to the object specified by o'.

For example, the rule representing the transition from the state *ReceivedOrder* to *DriveThroughLife* in the state diagram of Fig. 4 is depicted in Fig. 7 where objects are denoted by rectangles, states by rectangles with rounded corners, and events by ellipses. The rule contains objects of class *Client* and *DriveThrough*. On the left-hand side of the graph transformation rule, the *DriveThrough* object is attached to the state *ReceivedOrder*. On the right-hand side, the state *DriveThroughLife* is attached to the *DriveThrough*. The *DriveThrough* on the left-hand side has a pointer to the event *serve*(c) of its event queue. Applying the rule, this event is deleted and the event *eat* is connected to the client on the right-hand side. The guard of the rule guarantees that the client is the first one in the client queue of the drive-through. The structure of this rule reflects the structure of the transition where *serve*(c) is the initiating event (left-hand side) and eat is the induced action (right-hand side). The rules of the remaining transitions are depicted in Fig. 8.

In general, the graph transformation rule for a transition $t = (s, e, g, o'.e', s')$ is constructed as indicated in Fig. 9 where $o1$ is an object of the class the state diagram containing t is associated with, and $o2$ is the object specified by o'. The guard g of the rule must be checked before its application.

4 Integrated Specifications and Their Semantics

Class and state diagrams can be integrated in such a way that every class is connected with the state diagram describing its behavior. The integrated dynamic semantics can be obtained based on the combination of the graph transformation rules associated with the class diagram and the graph transformation rules associated with the state diagram.

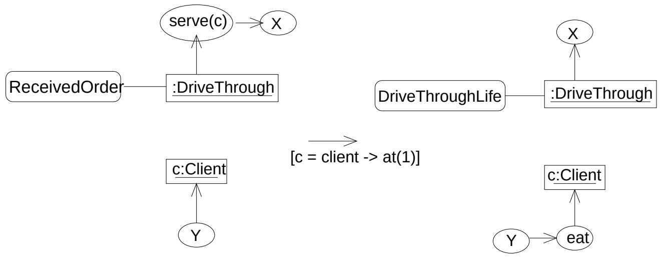

Fig. 7. The rule for the transition *serve* of Fig. 4

An *integrated diagram* is a pair $INTD = (CD, mstd)$ where CD is a class diagram and $mstd$ is a mapping assigning a state diagram $mstd(c)$ to every class c in CD such that $mstd(c)$ contains only events and call actions of class c.

Let *QUEUES* denote the class of *event queues*. Then a *system state* of an integrated diagram $(CD, mstd)$ is an object diagram of CD where every object is connected with a state of the state diagram associated with the class of the object. Moreover every object has a pointer to the first entry of its event queue. For our concrete example we will use event queues that work according to the FIFO principle. The queues are represented as string graphs where the nodes (denoted by ellipses) are labeled with events. For technical reasons there will also be a pointer from the last entry of every event queue to the object it belongs to. Moreover, empty event queues will be represented as loops attached to the corresponding object.

An example of a system state of the integrated diagram composed of the above class diagram and state diagrams is presented in Fig. 10.

The execution semantics of integrated diagrams is given by a set of graph transformation rules obtained from the combination of the rules presented in the previous section. Roughly speaking, the transition rules of state diagrams are glued with the rules of the classes they are associated with by identifying common objects and links. For example, the combination of the rule in Fig. 7 and the class diagram rule in Fig. 5 results in the graph transformation rule depicted in Fig. 11. The left-hand side of the shown integrated rule is obtained by gluing together the left-hand sides of the rules in Fig. 5 and 7 in such a way that the drive-throughs and the clients of both graphs are identified. To put it in another way, the combined left-hand side can be obtained by performing first the disjoint union of the left-hand sides of Fig. 5 and 7 and identifying then the drive-through of Fig. 5 with the drive-through of Fig. 7 and the client of Fig. 5 with the client of Fig. 7. The right-hand side of the combined rule is obtained by gluing the right-hand sides of the rules in Fig. 5 and 7 analogously. The resulting rule models the serving of a client c provided that the drive-through visited by c is in the state *ReceivedOrder* and has the event $serve(c)$ as dispatched event. The application of the rule deletes the link between the order (which has been submitted to the drive-through by the client) and the drive-through. Moreover,

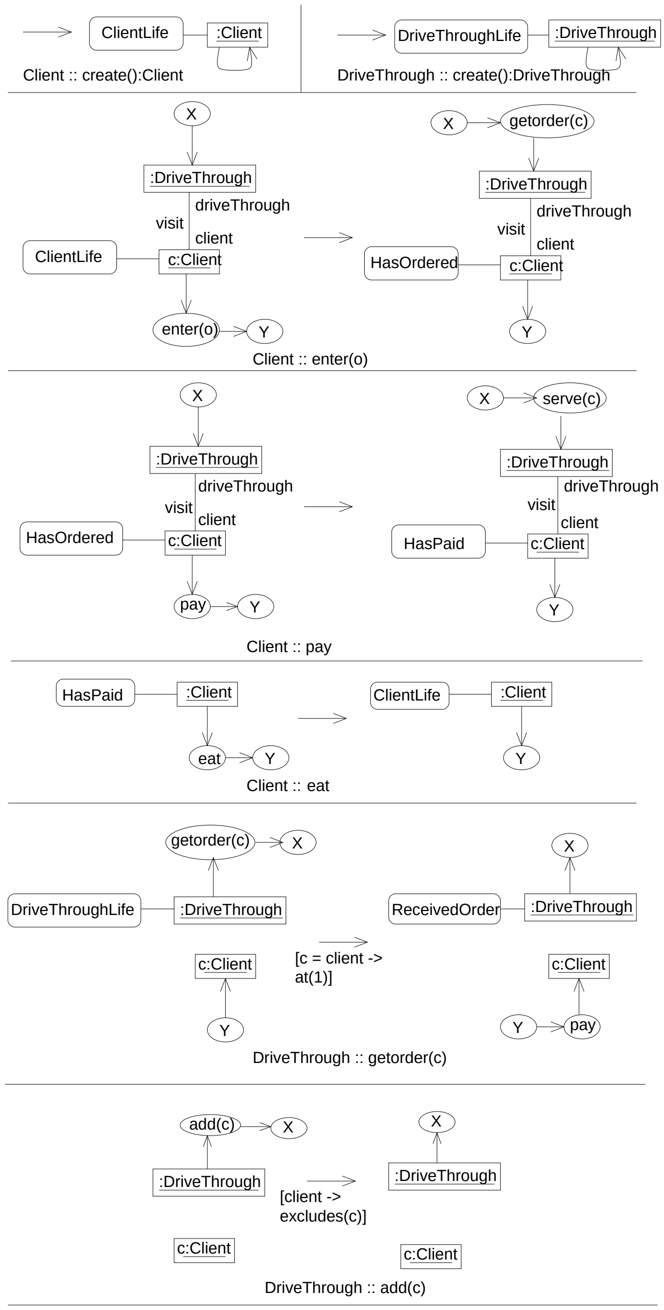

Fig. 8. Further rules for the transitions

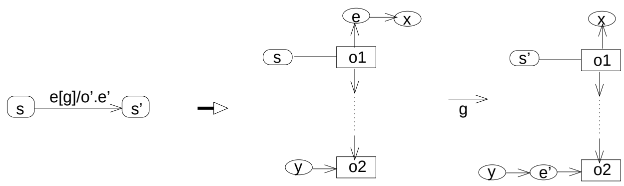

Fig. 9. The state changing rule schema

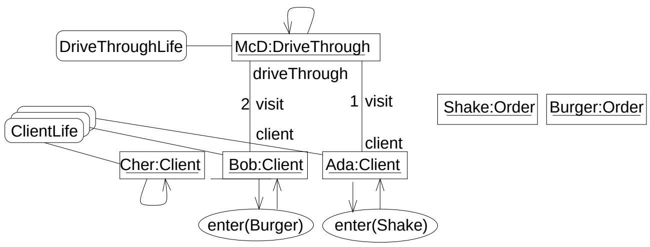

Fig. 10. An instance of an integrated diagram

it adds the event *eat* to the event queue of the client. Fig. 12 depicts further rules for our running example. The graph transformation rules which can be associated in the described way with an integrated diagram $INTD$ are called *integrated rules for* $INTD$.

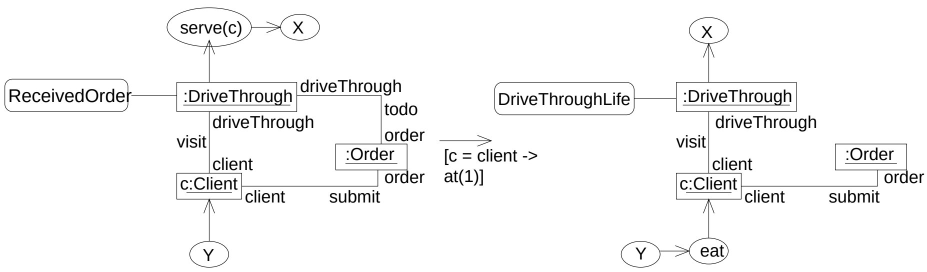

Fig. 11. The integrated rule for *serve*

An *integrated specification* is a triple $IS = (INTD, I, R)$ where $INTD$ is an integrated diagram, I is a system state of $INTD$, called the *initial system state*, and R is a set of integrated rules for $INTD$. The *semantics* of an integrated specification consists of all graphs which can be obtained from I by applying rules of R.

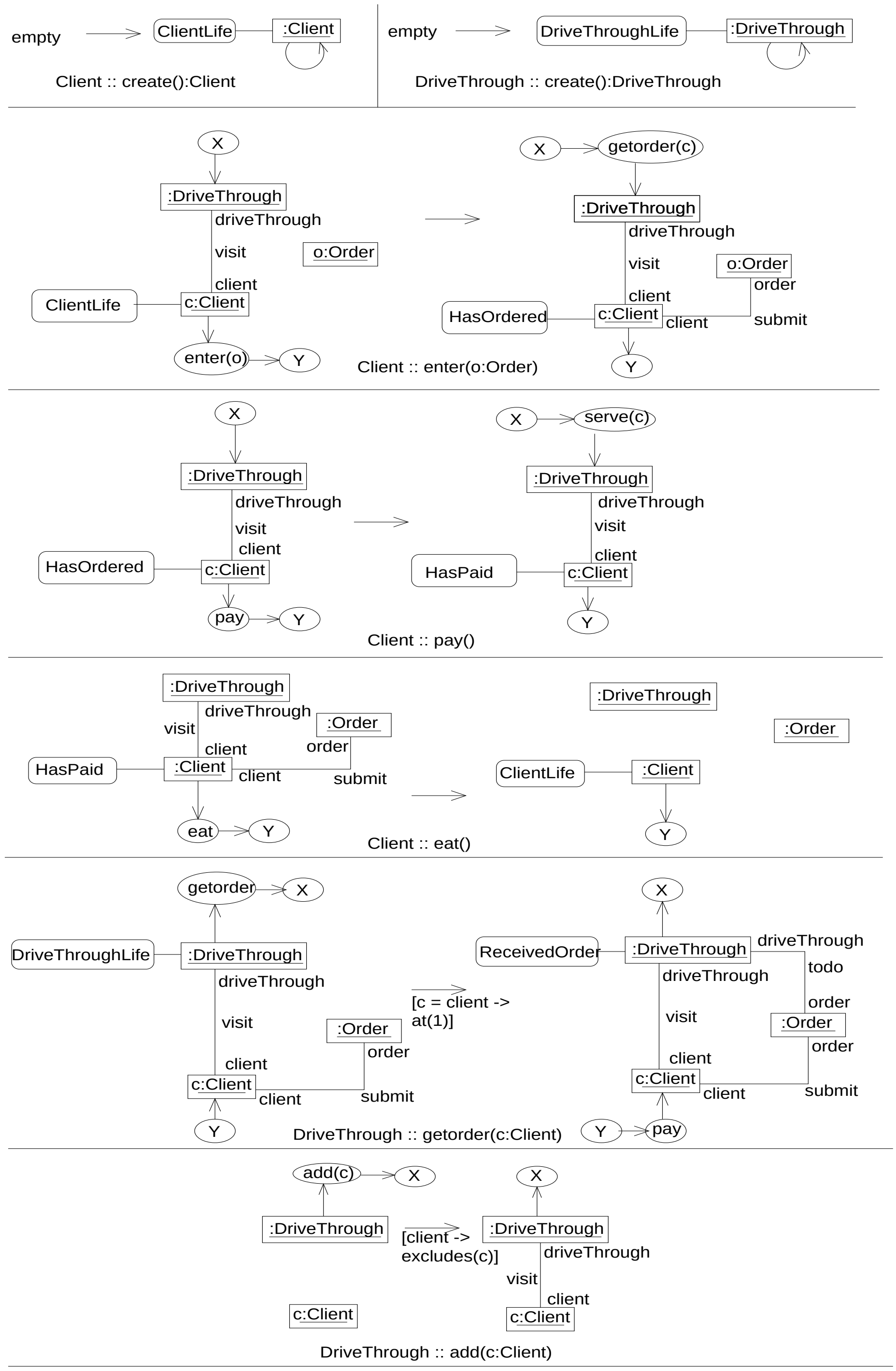

Fig. 12. Further integrated rules

An example of an integrated specification is $(INTD, I, R)$ where $INTD$ is composed of the class diagram in Fig. 1 and the state diagrams in Fig. 4 plus a state diagram for the class *Order* which just creates and destroys orders. The initial diagram is the empty graph and R consists of the rules presented in Fig. 12 plus some additional rules (for inserting events like $add(c)$ in event queues, or creating orders). Fig. 13 illustrates how the different system states (i.e. system states of integrated diagrams) can be derived with the example specification. The derivation starts with the integrated diagram of Fig. 10 and applies at first the rule *Ada.enter*(*Shake*). This means that after dispatching the event *enter*(*Shake*) a *submit*-link is created between *Shake* and *Ada*. Additionally, the event *enter*(*Shake*) is deleted from the event queue of *Ada*, *getorder*(*Ada*) is inserted in the event queue of *McD*, and the state of *Ada* changes from *ClientLife* to *HasOrdered*. Afterwards *McD.getorder*(*Ada*) is applied which changes the state of *McD* to *ReceivedOrder*, deletes *getorder*(*Ada*) from its event queue, and inserts *pay* in the event queue of *Ada*. Moreover, a *todo*-link is created between *McD* and *Shake*. Afterwards the rules *Ada.pay*(), *McD.serve*(*Ada*), *Ada.eat*(), and *Bob.enter*(*Burger*) are applied in this order.

5 Related Work

In the last years much research has been done in the formalization of UML semantics so that in the framework of this paper it is impossible to refer to all of them. Hence, in this section we mention only a selection of contributions on the formalization of UML semantics that are related to our work.

An approach to transform UML state diagrams into graphs to show the intended semantics is given in [GP98]. The expansion of nested states is explained by graph transformation. In [MP94] an operational semantics of statecharts is presented that is based on graph transformation. The work in [MSP96] describes formalization of semantics of specification languages by means of graph rewriting, exemplified for state diagrams. In [LP99] state machines are modeled as conditional term rewrite systems. [Kwo00] uses rewrite rules and operational semantics for model checking of UML state diagrams. Another approach for a formal operational semantics for UML state diagrams is discussed in [LMM99]. In [RACH00] a semantics for active classes with associated state machines is given that is based on labeled transition systems. In [BP01] it is illustrated how state diagrams can be automatically translated into high-level Petri nets by using graph transformation rules. Another approach employs graph transformation units [Kus00] to describe the execution semantics of state machines in a formal way [Kus01]. [EHHS00] proposes a graphical approach to the operational semantics of behavioral diagrams. In [EHK01] a methodology for the specification and verification of constraints for the UML is illustrated. The work in [EHHS00,EHK01,CEK01,EK99,FEL97] discusses the specification of UML semantics on the metamodel level.

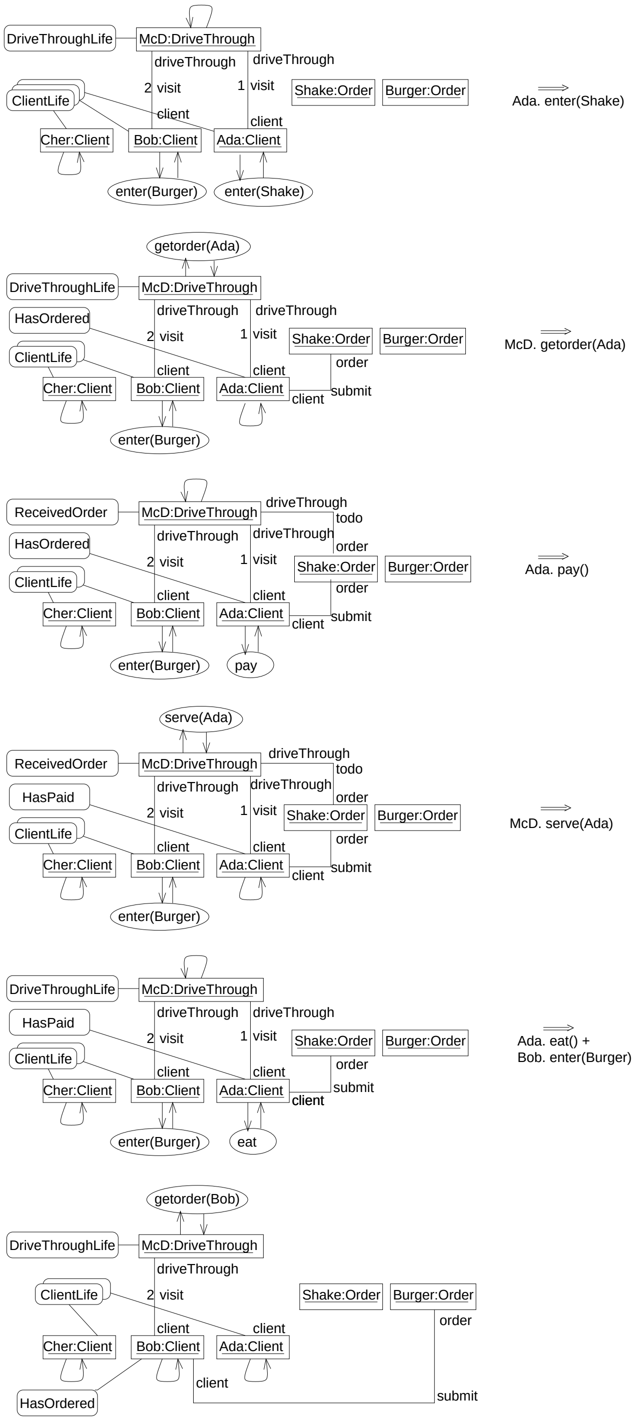

Fig. 13. A derivation

A graph-based semantics for OCL was developed by translating OCL constraints into expressions over graph rules [BKPPT00]. A visualization of OCL expression with collaboration diagrams can be found in [BKPT01].

[BD00] concentrates on verification aspects in the context of relational semantics. [LDSM01] proposes a general framework for integrating UML within formal techniques. [BPP99] discusses questions of making UML use cases more precise. [KLO+00] uses Catalysis concepts [DW98] for a comparison with UML. [FOW01] gives a CSP-like view on real-time systems described with UML.

6 Conclusion

We have introduced graph transformations for central language features of UML class and state diagrams. Operations from class diagrams and transitions from state diagrams have been described by single graph transformation rules, respectively. The single rules have mainly operated on UML object diagrams representing system states. These rules have been combined to yield a coherent single graph transformation system representing the combined integrated semantics of the class and state diagram.

Up to now we considered class, object, and state diagrams. We have designed our approach however in such a way that it is possible to handle UML interaction diagrams, i.e. sequence and collaboration diagrams, as well. In the most basic case, sequence and collaboration diagrams may be regarded as test cases which require consistency with the graph transformation system specified by the class and state diagrams.

There remain some open questions to be worked out in the future:

- The presented integrated semantics does not cover all language features of class and state diagrams. Hence it has to be investigated how other langauge elements can be handled.
- In complex cases the integration of various UML diagrams may lead to large diagrams which are dificult to handle and to understand. Therefore, for practical use, structuring concepts for graphs should be incorporated in the presented approach (cf. e.g. [Pra79,Har88,ES95,BEMW00,DHP02]).
- For being able to use our approach in practice, adequate transformation tools are needed. It should be investigated in which way existing tools can be employed to achieve this aim.
- A further interesting question is how far the presented integration approach can be used for consistency checking of different diagrams.

References

[BD00] Christie Bolton and Jim Davies. Using relational and behavioural semantics in the verification of object models. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000.

[BEMW00] Giorgio Busatto, Gregor Engels, Katharina Mehner, and Annika Wagner. A framework for adding packages to graph transformation systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 352–367, 2000.

[BKPPT00] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.

[BKPT01] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A visualization of ocl using collaborations. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 257–271, 2001.

[BP01] Luciano Baresi and Mauro Pezzè. On formalizing UML with high-level Petri Nets. In G. Agha and F. De Cindio, editors, *Proc. Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 271–300, 2001.

[BPP99] Ralph-Johan Back, Luigia Petre, and Ivan Porres Paltor. Analysing UML use cases as contracts. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 518–533. Springer, 1999.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[CEK01] Tony Clark, Andy Evans, and Stuart Kent. The metamodelling language calculus: Foundation semantics for UML. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th 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*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[DHP02] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 2002. To appear; short version in Proc. *FOSSACS 2000*, LNCS 1784.

[DW98] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, Singapore, 1999.

[EHHS00] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 – The Unified Modeling Language. Advancing the Standard*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337, 2000.

[EHK01] Gregor Engels, Reiko Heckel, and Jochen Malte Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In Martin

Gogolla and Cris Kobryn, editors, *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 272–286, 2001.

[EK99] Andy Evans and Stuart Kent. Core meta-modelling semantics of UML: The pUML approach. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 140–155. Springer, 1999.

[EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific, Singapore, 1999.

[ES95] Gregor Engels and Andy Schuerr. Encapsulated hierachical graphs, graph types, and meta types. In *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[FEL97] Robert France, Andy Evans, and Kevin Lano. The UML as a formal modeling notation. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Theory and Application to Graph Transformations (TAGT'98), Paderborn, November, 1998*, volume 1764 of *LNCS*. Springer, 1998.

[FOW01] Clemens Fischer, Ernst-Rüdiger Olderog, and Heike Wehrheim. A CSP view on UML-RT structure diagrams. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th 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*, volume 2029 of *LNCS*, pages 91–108. Springer, 2001.

[Gog00] Martin Gogolla. Graph transformations on the UML metamodel. In Jose D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and Joe B. Wells, editors, *Proc. ICALP Workshop Graph Transformations and Visual Modeling Techniques (GVMT'2000)*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000.

[GP98] Martin Gogolla and Francesco Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom Maibaum, and Bernhard Rumpe, editors, *Proc. ICSE'98 Workshop Precise Semantics of Modeling Techniques*, Technical Report TUM-I9803, pages 55–72, 1998.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[KK99] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[KLO+00] J. Küster Filipe, K.-K. Lau, M. Ornaghi, K. Taguchi, A. Wills, and H. Yatsu. Formal specification of Catalysis frameworks. In Jing Song Dong, Jifeng He, and Martin Purvis, editors, *Proc. 7th Asia-Pacific Software Engineering Conference*, pages 180–187. IEEE Computer Society Press, 2000.

[Kus00] Sabine Kuske. *Transformation Units—A structuring Principle for Graph Transformation Systems*. PhD thesis, University of Bremen, 2000.

[Kus01] Sabine Kuske. A formal semantics of uml state machines based on structured graph transformation. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 – The Unified Modeling Language. Modeling languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256, 2001.

[Kwo00] Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.

[LDSM01] Jing Liu, Jin Song Dong, Kun Shi, and Brendan Mahony. Linking UML with integrated formal techniques. In Keng Siau and Terry Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 13, pages 210–223. Idea Publishing Group, 2001.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*. Kluwer, 1999.

[LP99] Johan Lilius and Ivan Porres Paltor. Formalising UML state machines for model checking. In Robert France and Bernhard Rumpe, editors, *Proc. UML'99 – The Unified Modeling Language. Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445, 1999.

[MP94] Andrea Maggiolo-Schettini and Adriano Peron. Semantics of full statecharts based on graph rewriting. In Hans-Jürgen Schneider and Hartmut Ehrig, editors, *Proc. Graph Transformation in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 265–279, 1994.

[MSP96] Andrea Maggiolo-Schettini and Adriano Peron. A Graph Rewriting Framework for Statecharts Semantics. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073, pages 107–121. Springer-Verlag, 1996.

[OMG01] OMG. Unified Modeling Language specifcation, 2001. Available at http://www.omg.org//.

[Pra79] Terrence W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 389–400, 1979.

[PS00] Dorina C. Petriu and Yimei Sun. Consistent behaviour representation in activity and sequence diagrams. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. UML 2000 – The Unified Modeling Language. Advancing the Standard*, volume 1939 of *Lecture Notes in Computer Science*, pages 359–368, 2000.

[RACH00] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In Tom Maibaum, editor, *Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany*, volume 1783 of *Lecture Notes in Computer Science*, 2000.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, Singapore, 1997.

[SW00] Andy Schürr and Andreas J. Winter. UML packages for PROgrammed Graph REwriting Systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 396–409, 2000.

[TE00] Aliki Tsiolakis and Hartmut Ehrig. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In Hartmut Ehrig and Gabriele Taentzer, editors, *Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, Report Nr. 2000-2, Technical University of Berlin, pages 77–86, 2000.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.