Softw Syst Model (2009) 8:403–422
DOI 10.1007/s10270-008-0101-4

REGULAR PAPER

Towards an integrated graph-based semantics for UML

Sabine Kuske · Martin Gogolla ·
Hans-Jörg Kreowski · Paul Ziemann

Received: 3 August 2005 / Revised: 4 July 2008 / Accepted: 29 July 2008 / Published online: 28 August 2008

Abstract This paper shows how a central part of the Unified Modeling Language (UML) can be integrated into a single visual semantic model. It discusses UML models composed of class, object, state, sequence and collaboration diagrams and presents an integrated semantics of these models. As formal basis the theoretically well-founded area of graph transformation is employed which supports a visual and rule-based transformation of UML model states. For the translation of a UML model into a graph transformation system the operations in class diagrams and the transitions in state diagrams are associated with graph transformation rules that are then combined into one system in order to obtain a single coherent semantic description. Operation calls in sequence and collaboration diagrams can be associated with applications of graph transformation rules in the constructed graph transformation system so that valid sequence and collaboration diagrams correspond to derivations, i.e., to sequences of graph transformation rule applications. The main aim of this paper is to provide a formal framework that supports visual simulation of integrated UML specifications in which system states and state changes are modeled in a straightforward way.

Keywords UML diagram · Graph transformation · Formal semantics

Communicated by Prof. Robert France.

Sabine Kuske and Hans-Jörg Kreowski would like to acknowledge that their research is partially supported by the Collaborative Research Centre 637 (Autonomous cooperating logistic processes: a paradigm shift and its limitations) funded by the German Research Foundation (DFG).

S. Kuske (✉) · M. Gogolla · H.-J. Kreowski · P. Ziemann
Department of Computer Science,
University of Bremen, P.O. Box 330440, 28334 Bremen, Germany
e-mail: kuske@informatik.uni-bremen.de

M. Gogolla
e-mail: gogolla@informatik.uni-bremen.de

H.-J. Kreowski
e-mail: kreo@informatik.uni-bremen.de

1 Introduction

In recent years, the Unified Modeling Language (UML) [4,34,41] has been widely accepted as a standard language for modeling and documenting software systems. The UML offers a number of diagram types that can be used to describe particular aspects of software artifacts. These diagram types can be divided depending on whether they are intended to describe structural or behavioral aspects. From a fundamental point of view, one meaningful way of employing UML is to use class, state and interaction diagrams as the basic means for system description, because class diagrams determine the fundamental object structures, state diagrams can be employed for describing the fundamental object behavior, and interaction diagrams serve to specify how objects interact in a collaboration.

Unfortunately, UML diagrams were introduced without a formal semantics that maps the diagrams to a mathematically precise semantic domain. Their interplay within a UML model is neither formally defined, i.e., even if one has a semantics for evey diagram type, it is still not clear how to get an integrated formal semantics for the whole UML model.

A lot of research has been done in recent years to formalize single parts of UML. However, defining a formal semantics for the UML as a whole is complex due to the vast scope of the UML. In this paper we present a first step towards an integrated formal semantics of UML, which takes into account five basic diagram types, namely class, object, state, sequence, and collaboration diagrams. The presented semantics is related to UML 1 but the concepts considered here are also contained in UML 2 where collaboration diagrams are called communication diagrams.

For the formalization of an integrated semantics of UML models we employ graph transformation [9,11,40], which is a well-developed field and has many application domains, such as graphical modeling languages like the UML. The main part of a graph transformation system is a set of graph transformation rules that successively transform local parts of graphs. In general, graph transformation is very adequate to formalize and visualize system behavior because system states can be represented as graphs and system execution steps as applications of graph transformation rules. In particular, the possibility of visualizing complex interconnections as graphs and the rule-basedness of graph transformation establishes a tight connection to some fundamental features of UML:

1. System states in UML can be represented as object diagrams, which in turn can be formalized as graphs.
2. System behavior can be described in UML with state diagrams in which each transition corresponds to an atomic system evolution step. Since system states are graphs, the firing of a transition can be represented as the application of a graph transformation rule.
3. Sequences of such atomic system evolution steps can be described by UML interaction diagrams, i.e., sequence and collaboration diagrams. This means that interaction diagrams can be translated into sequences of graph transformation steps.
4. Graphs can be understood as visual entities like all diagrams in the UML. Explaining UML by graph transformation means to close only a small gap between the language to be defined, namely the UML, and the language used as the semantic target language, namely a set of graphs.

We do not claim that graph transformation is the only possible framework for a formal intergrated UML semantics, but it is well-tried, general and flexible enough. And as graphs and diagrams are closely related to each other, the intuition behind UML is not lost. Other formalizations of the semantics of UML diagrams rely for example on Petri nets [1], term rewriting [29,30], labeled transition systems [2,35], temporal logic [39], set theory [43], or OCL [38]. Apart from [1] these approaches focus on the semantics of one or two diagram types but not on an integrated semantics for UML. Moreover, the underlying theories do not support the visualization of system states and system behavior in the described straightforward way.

In the integrated formal semantics of this paper, class, object and state diagrams are mapped into a graph transformation system, sequence and collaboration diagrams into transformations performed by the system. Table 1 shows the notions from UML that we use and the corresponding notions in the area of graph transformations.

The aim of the presented integrated formal semantics of UML is to get a solid basis for main research topics like validation, verification and syntax checking. This means that the representation of a UML model as a graph transformation system facilitates the validation of the system by comparing transformed system states with the expectations of the modeler. Furthermore, the theory of graph transformation can be used to verify properties of UML models, for example to check whether an interaction (i.e., a graph transformation) can only occur in a certain set of system states. Finally, syntactically incorrect diagrams can be discarded if they cannot be formalized as graphs or graph transformation rules.

To keep the technicalities feasible and to avoid overloading, we do not attempt to cover the whole of UML in this first major step towards an integrated semantics. For example, we do not consider UML interaction diagrams for operation specification but assume UML state diagrams and proper graph transformational specifications instead. These graph transformation rules can be regarded as UML ≪become≫ flow relationships between object diagrams. Moreover, we consider only simplified diagram types that do not cover concepts like inheritance, composite states, etc. The missing features will be integrated in further steps.

The proposed integrated semantics of UML is not meant as the ultimate answer to all questions, but as one possibility that realizes the intuitive meaning and behavior of the considered diagrams and their interplay in a reasonable way. Alternatives and variations are thinkable. If they would be formalized in

Table 1 UML and graph transformation notions

UML notion	Notion in the graph transformation approach
Class diagram	Set of system states represented by graphs and a set of graph transformation rules as semantics for operations
Object diagram	System state
State diagram	Graph transformation rules transforming system states into system states
Sequence diagram	Derivation in the defined graph transformation system
Collaboration diagram	Derivation in the defined graph transformation system

the framework of graph transformation, too, then one would have the chance to formulate the differences formally and to prove them.

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 explains how class and state diagrams can be translated into graphs and transformation rules. Section 4 shows how the graphs and the graph transformation rules resulting from class and state diagrams can be integrated into a single graph transformation system. Section 5 describes the relationship between sequence and collaboration diagrams and the respective derivations of the graph transformation system. It is sketched how these derivations can help to check whether the model is adequate, for example, to check whether a given message sequence is applicable in a certain system state. All concepts are illustrated by a single running example. Section 6 mentions related work. The paper closes in with some final remarks.

Two preliminary versions of this paper are [28] and [20]. The former focuses on integrating class, object, and state diagrams whereas the latter considers also the integration of interaction diagrams.

2 Class, object, and state diagrams

Class, object, and state diagrams are fundamental diagrams of the UML. In the following we briefly illustrate these diagram types. As already mentioned, in this approach towards an integrated UML semantics we consider simplified versions of UML diagrams. For further details concerning UML diagrams, the reader is referred to, e.g., [4,34,41].

2.1 Class diagrams

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. Every class c specifies a set of objects called the *instances of* c. An *association end* is a language element of class diagrams which connects 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 represent classes, and the edges represent associations, generalizations, or dependencies. We concentrate here on binary associations only. Some of the classes may be associations as well. These classes are called association classes.

Figure 1 shows an example of a class diagram consisting of classes and binary associations where association names and roles are omitted. It models an office containing six

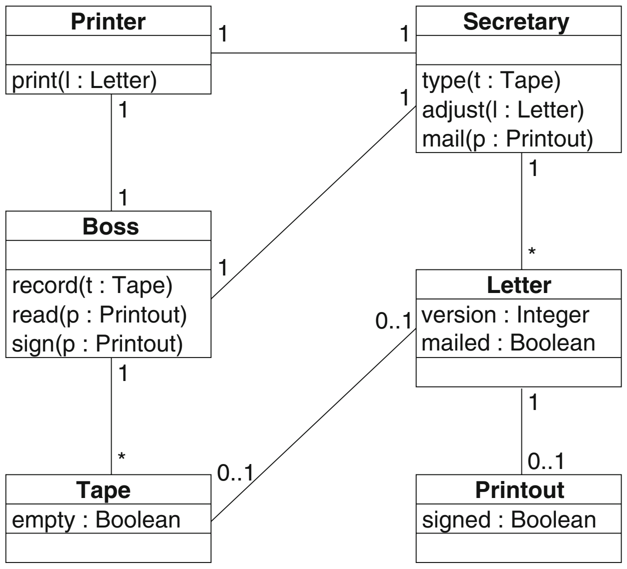


Fig. 1 A class diagram

classes, namely *Printer*, *Secretary*, *Boss*, *Letter*, *Tape*, and *Printout*. Some of the classes contain operations which describe the actions an object of the class is able to perform. For example, a secretary can type a letter which is recorded on a tape or mail a printout. In the diagram there are also some classes with attributes. For example, a tape can be empty or not which is indicated by the boolean value of the attribute *empty* of the class *Tape*. The associations of the class diagram connect different classes and contain multiplicities that prescribe the number of objects that can be linked to each other. For example, one printer can be used by one secretary and one secretary can use one printer. Analogously, one boss has one secretary, one printer, and arbitrarily many tapes.

Class diagrams can be formally defined as directed labeled graphs. Let A and B be alphabets. Then a *directed labeled graph* over (A, B) is a system $G = (V, E, s, t, l, m)$ where

- V is a finite set of *nodes*;
- E is a finite set of *edges*;
- $s, t, : E \to V$ assign a *source node* $s(e)$ and a *target node* $t(e)$ to every $e \in E$;
- $l: V \to A$ assigns a *node label* $l(v)$ to every node v in V; and
- $m: E \to B$ assigns an *edge label* $m(e)$ to every edge $e \in E$.

The components of V, E, s, t, l, and m are also denoted by V_G, E_G, s_G, t_G, l_G, and m_G, respectively.

In a class diagram, every node is labeled with a class name, and every edge with a triple consisting of an association name, a pair of roles and a pair of multiplicities. Let $\mathscr{C}$ be a set of classes, let $\mathscr{R}$ be a set of roles, let $\mathscr{A}$ be a set of association names, and let $\mathscr{M}$ be a set of multiplicity specifications, i.e., every $x \in M$ specifies a set $\mathrm{SEM}(x) \subseteq \mathbb{N}$. Then a *class diagram* is a directed labeled graph over $(\mathscr{C}, \mathscr{A} \times \mathscr{R}^2 \times \mathscr{M}^2)$. For every $e \in E$ with $m(e) = (a, r_1, r_2, x_1, x_2)$, the triple (a, r_1, r_2) is called the *names* of e denoted by $names(e)$, x_1 is the *source multiplicity* of e denoted by $sm(e)$ and x_2 is the *target multiplicity* of e denoted by $tm(e)$.

We do not consider class inheritance yet but we believe that this important concept of object orientation can be integrated in a further step. Inheritance may also be resolved via delegation, as pointed out for example in [19]. As mentioned earlier, class diagrams may have aggregations and compositions which are special cases of associations. They represent relations between a *whole* and a *part*. Additionally, in the case of compositions, the lifetime of every object depends on the lifetime of the object which it is a part of. For example, in the class diagram of Fig. 1, compositions could be used to express that every printer is a part of either a secretary or a boss. Hence, with this solution, every secretary as well as every boss has her/his own printer. If a class diagram CD contains aggregations or compositions, the system states represented by CD must satisfy certain requirements. For example, chains of objects related by instances of aggregations or compositions are not allowed to be cyclic. Class diagrams with aggregation and composition could be formally defined as above, but where every edge has an additional label indicating whether it represents an association, a composition or an aggregation. Moreover, the multiplicity of the whole in a composition must be equal to $\{0\}$. In [19] it is shown that aggregation and composition can be equivalently substituted by simple associations with additional OCL constraints, which have to be valid in each system state.

In the above definition of class diagrams, binary associations are represented by directed edges that have the navigation direction of the represented association. Hence associations with a bi-directional navigability can be represented by two directed edges with the same label but pointing in opposite directions. If one additionally allows the use of hyperedges instead of only binary edges, class diagrams with n-ary associations can be defined as directed labeled hypergraphs in a straighforward way, so that, in particular, association classes could be modeled as a special kind of ternary hyperedges (cf. [18]).

Please note that throughout this paper diagrams are depicted and defined in a concrete syntax making it comprehensible for the reader. However, for the detailed formalization of our approach, especially for an implementation of it, they are represented in a more abstract way, e.g., as instances of a meta-model (cf. [49]).

2.2 Object diagrams

Object diagrams differ from class diagrams in the sense that they contain objects instead of classes and links instead of associations. Object diagrams are useful to represent the 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 a printer *lw1*, a boss *ada*, a tape *t* which is not empty, and a secretary *sam*. The printer can be used by *sam* and *ada* and *t* is the tape of *ada*.

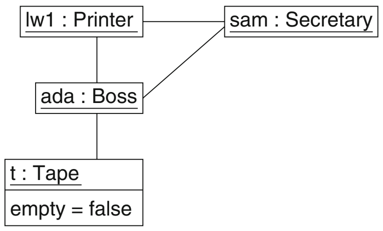


Fig. 2 An object diagram

Object diagrams can be formally defined as directed labeled graphs where the nodes are labeled with objects and the edges represent links. For each class $c \in \mathscr{C}$, let $\mathscr{O}(c)$ denote the set of all instances of c. Then an object diagram is a directed labeled graph over $(\bigcup_{c \in \mathscr{C}} \mathscr{O}(c), \mathscr{A} \times \mathscr{R}^2)$.

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 are satisfied. For example, the object diagram of Fig. 2 fits the class diagram in Fig. 1 but drawing an additional link from *sam* to *t* would violate the requirement that links must be instances of associations.

Formally, an object diagram $\mathrm{OD} = (V, E, s, t, l, m)$ *fits* a class diagram $\mathrm{CD} = (V', E', s', t', l', m')$ if there exist two mappings $g_V : V \to V'$ and $g_E : E \to E'$ such that the following holds.

- For every $v \in V$, $l(v) \in \mathscr{O}(l'(g_V(v)))$, i.e., every object node $v \in V$ that is mapped to a node $v' \in V'$ must be labeled with an object of the class, v' is labeled with.
- For every $e \in E$, $g_V(s(e)) = s'(g_E(e))$ and $g_V(t(e)) = t'(g_E(e))$, i.e., the mappings are structure preserving.
- For every $e \in E$, $m(e) = \mathrm{names}(g_E(e))$, i.e., the label of every link edge e is equal to the names of the association to which e is mapped by g_E.
- For every $e' \in E'$, $|\{s(e) \mid g_E(e) = e'\}| \in \mathrm{SEM}(sm(e'))$ and $|\{t(e) \mid g_E(e) = e'\}| \in \mathrm{SEM}(tm(e'))$, i.e., the number of sources of the same kind of links is contained in the source multiplicity specification of the association to which the links are mapped. Analogously, the number of targets of the links must be specified by the target multiplicity specification of the related association.
- If the class diagram CD contains also aggregations or compositions, the object diagram OD must satisfy some additional requirements such as the cycle-freeness of

some link chains. These requirements can be expressed by OCL expressions [19,38].

2.3 State diagrams

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 one of which is an initial state and a set of transitions connecting states. 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 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 *node* 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 [34,48], i.e., a logic formula that evaluates either to *true* or to *false* and has no side effects. A transition can only fire if its guard evaluates to *true*. Guards can be used to check whether some attributes satisfy certain requirements. For example, we could require that the operation *sign(p: Printout)* of the class *Boss* can only be executed if the attribute *signed* of the parameter p is equal to *false*. Another example of using guards is to require that some objects of the current system state be in a certain state. Guards are often used to obtain deterministic state machines: If there are several transitions with the same source but different targets, and the firing of all these transitions means to execute the same operation, mutual exclusive guards can be employed to guarantee that only one of the transitions can fire. Events and actions can be of many types [41]. In the following we restrict ourselves to call events and a simple kind of call actions. A (*call*) *event* represents the dispatch of an operation and may trigger the firing of the transition it is labeled with. The object which receives the event executes the corresponding operation. An event is of the form $\mathrm{op}(p_1, \ldots, p_n)$ where op is the name of an operation and $p_1, \ldots, p_n$ are parameters. A (*call*) *action* invokes an operation of an object and is of the form $o.\mathrm{op}(p_1, \ldots, p_n)$ where o is a path in the class diagram from the class the state diagram is associated with, say c, to some class, say c', of which op is an operation. In general, such a path can be defined as an alternating sequence of classes and associations, where the first component is the class c and the last component is the class c'. This situation is illustrated in Fig. 3, which shows a class diagram and a state diagram. The class diagram contains the class c the state diagram is associated with. The state diagram shows a transition with call action $o.\mathrm{op}(p_1, \ldots, p_n)$. The path o goes from c to the class c', which is indicated by the dashed arrow from c to c'. Class c' contains an operation op which is called for some object of class c' with the parameters $p_1, \ldots, p_n$ when the transition fires.

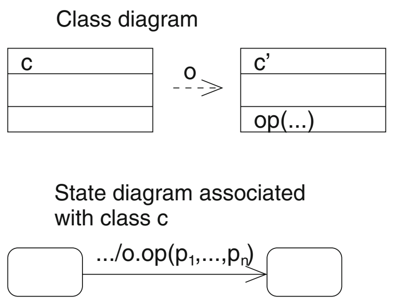


Fig. 3 Illustration of a call action

Consider for example the transition

type(t)/*self*.*printer*.*print*$(t.$*letter*$)$

of the state diagram for the class *Secretary* in Fig. 4. The call event of this transition is *type(t)*. The path corresponds to *self.printer*, which is the path from the class *Secretary* to the class *Printer*. The operation to be called for some object of class *Printer* is *print(t.letter)*.

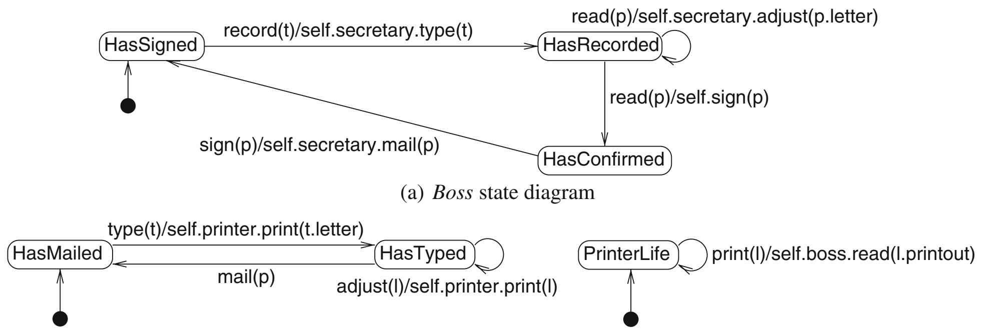
A state diagram can be formally defined as a system STD= $(S, \mathscr{E} \times \mathscr{G} \times \mathrm{Act}, d, s_0)$ where S is a finite set of *states*, $\mathscr{E}$ is a set of *events*, $\mathscr{G}$ is a set of *guards*, Act is a set of *actions*, $d \subseteq S \times (\mathscr{E} \times \mathscr{G} \times \mathrm{Act}) \times S$ is a finite set of *transitions*, and $s_0 \in S$ is the *initial state*. The class of all state diagrams is denoted by SD.

In Fig. 4, the state diagrams for the classes *Boss*, *Secretary*, and *Printer* are depicted. The objects of class *Boss* can be in the state *HasSigned*, *HasRecorded* or *HasConfirmed*. An object of the class *Secretary* can be in the states *HasMailed* or *HasTyped*. Finally, a printer has only the state *PrinterLife*.

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 in a given class diagram CD. Let $t = (s, e, g, o.\mathrm{op}, s')$ be a transition in STD with source state s, target state s', event e, guard g, and call action $o.\mathrm{op}$. The firing of a transition takes place in an object diagram that fits the class diagram CD. Every path p of the object diagram can be mapped to a path $g(p)$ in CD by restricting the domain of the mappings g_V and g_E of Sect. 2.2 to the objects and links of p. In this case we say that p is an *instantiation* of the path $g(p)$. Given an object diagram that fits the underlying class diagram CD the firing of transition $t = (s, e, g, o.\mathrm{op}, s')$ comprises the following steps.

1. Check whether the following conditions are satisfied. (a) There is an object x of class c that is in state s.[1] (b) The

[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 boss is initially in the state *HasSigned*, every secretary in the state *HasMailed* and every printer in the state *PrinterLife*.


HasSigned
record(t)/self.secretary.type(t)
HasRecorded
read(p)/self.secretary.adjust(p.letter)
read(p)/self.sign(p)
sign(p)/self.secretary.mail(p)
HasConfirmed
HasMailed
type(t)/self.printer.print(t.letter)
HasTyped
mail(p)
adjust(l)/self.printer.print(l)
PrinterLife
print(l)/self.boss.read(l.printout)

(a) *Boss* state diagram

(b) *Secretary* state diagram

(c) *Printer* state diagram

Fig. 4 State diagrams for *Boss*, *Secretary*, and *Printer*

next event in the event queue of x is equal to e. (We assume that every object has an associated event queue.) (c) The guard g can be evaluated to *true*. (d) The path o has an instantiation in the object diagram being a path from x to another object y.

2. Execute the call event e, send the operation call op to the object y, and change the state of x from s to s'. The sending of an operation call to y corresponds to its insertion in the event queue of y.

As an example consider the state diagrams in Fig. 4. In the state diagram of the class *Boss* the transition from the state *HasSigned* to state *HasRecorded* can fire if there is an object x of class *Boss* such that x is in state *HasSigned*, the next event to be dispatched from the event queue of x is *record*(t), and there is a link from x to some *secretary* object y. When the transition fires the operation *type*(t) of the secretary of the boss x is called. This means that the event *type*(t) is written into the event queue of the secretary.

The office process modeled by the three state diagrams in Fig. 4 is as follows: A boss takes a dictation of a letter on tape, then gives it to her or his secretary for typing it. The secretary calls the printer to print the letter. The boss reads the printout and then either signs it and tells the secretary to mail the letter or asks the secretary to adjust it. After adjusting, the letter is printed by the printer and read by the boss again. Possible sequences of events for the office process can be specified in UML with sequence or collaboration diagrams. Examples are given in Sect. 5 later by the diagrams in Figs. 16 and 17.

For technical simplicity, we assume that the parameters of call events are objects and that every parameter in a call action is an object or a path to an object. Consider for example the transition *type(t)/self.printer.print(t.letter)* of the state diagram for the class *Secretary*. The parameter t is an object of type *Tape* and the parameter *t.letter* specifies the letter linked to tape t. This assumption allows to represent parameters visually as objects which can be transformed via graph transformation rules. It is worth noting, that this assumption does no harm because data types can be represented as classes in a natural way.

Run-to-completion steps can be formally described by graph transformation rules. Sections 3 and 4 show how transitions can be translated into graph transformation rules such that the firing of a transtition corresponds to an application of the rule.

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 application domains (see [9,11,40] for an overview). In the following we briefly present the basic concepts of graph transformation.

3.1 Graph transformation

The basic operation of graph transformation comprises the local manipulation of graphs via the application of a rule. A *graph* consists of a set of (attributed) nodes and a set of (attributed) edges. Examples of graphs are the class diagram and the object diagram presented in the previous section where the nodes represent classes and objects, and the edges associations and links, respectively.

A *graph transformation rule* mainly consists of two graphs, called *left-hand side* and *right-hand side* which have a common part. The left-hand side and the right-hand side are object diagrams. The common part of the left- and right-hand side is the set of all nodes and edges that are contained in both sides. A rule with left-hand-side L and right-hand-side R is depicted as $L \rightarrow R$ where the common nodes and edges of L and R have the same relative position in the left- and the right-hand side. The parts of the sides that do not belong to the common part are exposed by bold lines and face. An example of a rule is depicted in Fig. 5. The common part of the rule is equal to its left-hand side which consists of a secretary and a tape which is not empty.

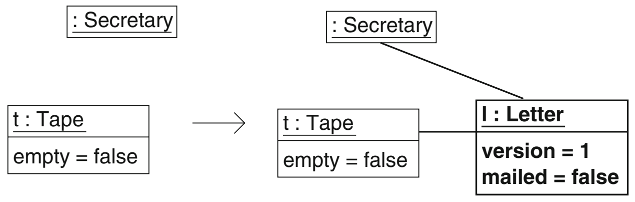


Fig. 5 Rule for *Secretary*::*type*(t)

For defining graph transformation rules, the notion of a subgraph is needed. A graph G is a *subgraph* of a graph H, denoted by $G \subseteq H$, if $V_G \subseteq V_H$, $E_G \subseteq E_H$, and the inclusions are structure-preserving, i.e., $s_G(e) = s_H(e)$, $t_G(e) = t_H(e)$, $m_G(e) = m_H(e)$ for all $e \in E_G$, and $l_G(v) = l_H(v)$ for all $v \in V_G$. Now a *graph transformation rule* can be defined as a triple $r = (L, K, R)$ of graphs such that $L \supseteq K \subseteq R$.

A rule (L, K, 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 image of the common part K is maintained. The application of the rule in Fig. 5 adds a letter l to a diagram in which the left-hand side occurs, and it adds a link between l and the tape and a link between l and the secretary. The rule can be applied to the object diagram of Fig. 2. (In our example nothing is specified concerning the contents of the letter l and the tape t. In order to guarantee their equality one could add a further attribute with the contents of the letter resp. the tape and require that they are equal if they are linked together.)

For defining rule application formally, we need the definition of a *graph morphism* $g: G \to H$ where G and H are graphs. Each such morphism consists of a pair (g_E, g_V) of mappings such that $g_E: E_G \to E_H$ and $g_V: V_G \to V_H$ satisfy the following.

- For every $e \in E$, $g_V(s_G(e)) = s_H(g_E(e))$ and $g_V(t_G(e)) = t_H(g_E(e))$, i.e., the mappings are structure preserving.
- For every $v \in V_G$, $l_G(v) = l_H(g_V(v))$, i.e., the mapping g_V preserves node labels.
- For every $e \in E_G$, $m_G(e) = m_H(g_E(e))$, i.e., the mapping g_E preserves edge labels.

The graphs G and H are called *isomorphic*, denoted by $G \cong H$, if g_V and g_E are bijections. The image of the graph G in H is denoted by $g(G)$, and for subsets $E \subseteq E_G$ and $V \subseteq V_G$, the set of images of E and V are denoted by $g_E(E)$ and $g_V(V)$, respectively.

The *application* of a rule $r = (L, K, R)$ to a graph G yields a graph G' if G' can be obtained as follows:

1. Choose a graph morphism $g: L \to G$.
2. Check the *contact condition* that avoids dangling edges during the application process: If the image of a node $v \in V_L$ is the source or the target of an edge not in the image of L (i.e., $g_V(v) = s_G(e)$ or $g_V(v) = t_G(e)$ for some edge $e \in E_G - E_{g(L)}$), then v must be in K, i.e., it cannot be deleted during the application of the rule.
3. Check the *identification condition* that prescribes that only items of K can be identified via g, i.e., for all $v, v' \in V_L$ with $g_V(v) = g_V(v')$ it is required that $v, v' \in V_K$; analogously for edges.
4. Construct the *intermediate graph* D by deleting from G the edges and nodes in $g(L)$ up to the items in $g(K)$, i.e., $E_D = E_G - g_E(E_L - E_K)$, $V_D = V_G - g_V(V_L - V_K)$, and s_D, t_G, l_G, and m_G are restrictions of s_G, t_G, l_G, and m_G, respectively so that $D \subseteq G$.
5. Glue R and D in K by identifying all items in K with their images, i.e., construct a graph that is isomorphic to G' where $V_{G'} = V_D \uplus (V_R - V_K)$, $E_{G'} = E_D \uplus (E_R - E_K)$,[2]

$$s_{G'}(e) = \begin{cases} s_R(e) & \text{if } e \in E_R - E_K \text{ and} \\ & s_R(e) \in V_R - V_K \\ g_V(s_R(e)) & \text{if } e \in E_R - E_K \text{ and} \\ & s_R(e) \in V_K \\ s_D(e) & \text{otherwise,} \end{cases}$$

$t_{G'}$ is defined analogously to $s_{G'}$, $l_{G'}(v) = l_D(v)$ if $v \in V_D$, $l_{G'}(v) = l_R(v)$ if $v \in V_R - V_K$, $m_{G'}(e) = m_D(e)$ if $e \in E_D$, and $m_{G'}(e) = m_R(e)$ if $e \in E_R - E_K$.

The application of r to G yielding G' is denoted by $G \underset{r}{\Longrightarrow} G'$. The gluing of R and D in K corresponds to the construction of a pushout in the context of category theory. Moreover, since the gluing of L and D in K yields the graph G (or an isomporphic copy of G), the described approach to transform graphs is called double-pushout approach [8]. This is a central approach in the area of graph transformation; not only is it theoretically well-studied but it has also been successfully proposed as a formally well-founded modeling framework in many areas of computer science. Since we do not assume that every reader of this paper is familiar with category theory we decided to give a set-theoretical description of the approach.

The iterated application of graph transformation rules is called a *derivation*, denoted by $G \overset{*}{\underset{P}{\Longrightarrow}} G'$ where P is a set of rules from which the applied rules are taken, i.e., $G \overset{*}{\underset{P}{\Longrightarrow}} G'$ stands for all derivations $G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ with $G_0 \cong G$, $G_n \cong G'$, and $r_1, \ldots, r_n \subseteq P$. An example of a derivation is given later in Figs. 14 and 15.

[2] $\uplus$ denotes the disjoint union of sets.

In order to describe class operations in an adequate way, we allow attributes as node and edge labels [12,31]. On one hand, attributed graph transformation allows computations on labels of nodes and edges during the application of a graph transformation rule. On the other hand, attributes may contain parameters so that one rule can represent a set of concrete variable-free graph transformation rules. For example, the symbol t in the rule in Fig. 5 is a parameter that can be instantiated with any name of a tape. An example for a rule that computes on attributes will be given in Fig. 6.

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 Sect. 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.

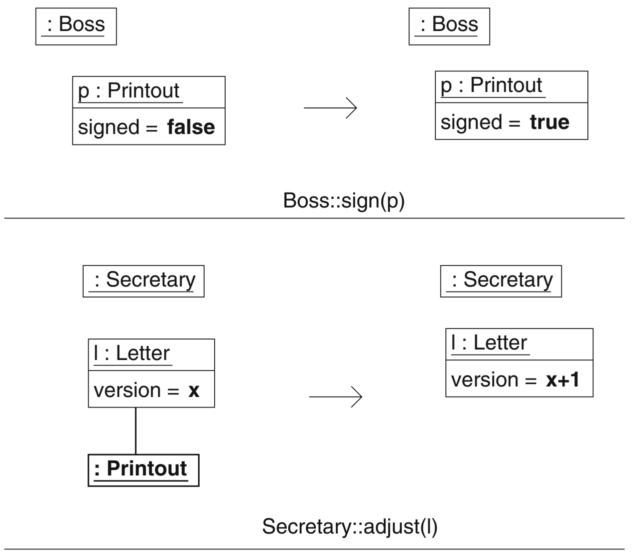


Fig. 6 Further rules for the class diagram of Fig. 1

3.2 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. Each such 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 $record(t)$ of the class *Boss* applied to the object diagram of Fig. 2 changes the value *empty* of t from *true* to *false*. The rule $type(t)$ of Fig. 5 can then be applied to the resulting object diagram. Analogously we can assign a graph transformation rule to every other operation of our example class diagram.

In the following we require that every rule r that models an operation of a class c contain in the common part a (parameterized) object node of type c that represents the object that executes the operation. This object node will be denoted by mainobject(r). Please note that this requirement is meaningful because it guarantees that only existing objects can execute operations.

Since the rule $Boss::read(p)$ does not change the object diagram, all three parts of the rule just consist of the main object, namely a node of class *Boss*. The rule for the operation $sign(p)$ of class *Boss* changes the attribute *signed* of a printout from *false* to *true*. The rule $Secretary::adjust(l)$ removes a printout of a letter and increases the version number of the letter by one. These two rules are shown in Fig. 6. The rules for the remaining operations can also be described with graph transformation rules.

Graph transformation rules provide a means which allows 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 preconditions for the execution of the operations by adding requirements like objects with specific attribute values or links into the left-hand side. Hence, we require that effect of the execution of class operations is given by graph transformation rules.

Given a graph transformation rule, it can be checked automatically whether the application of a class operation specified as a graph transformation rule yields a valid object diagram, i.e., an object diagram fitting the underlying class diagram CD. On the one hand the graphs in the rules must fit the structure of CD but not the multiplicity constraints, i.e., for every graph G in a rule there must be mappings $g_V: V_G \to V_{\mathrm{CD}}$ and $g_E: E_G \to E_{\mathrm{CD}}$ that satisfy the first three of the requirements given in the definition of fitting objects. Clearly, this can be checked statically. On the other hand, before applying a rule it must be checked that the multiplicity constraints are not violated. This can be expressed via adequate application conditions (see also [10,21]). For example, the fact that no second secretary can be linked to the same boss can be expressed with the negative application condition that forbids the existence of a link from the boss to a secretary in the current object diagram.

3.3 Representing transitions as graph transformation rules

The transitions of a state diagram STD can also be represented by means of graph transformation rules. Let c be the class the state diagram STD is associated with, and let o_1 be an object of class c. Let $t = (s, e, g, o.e', s')$ be a transition of STD where—as before – s denotes the source state

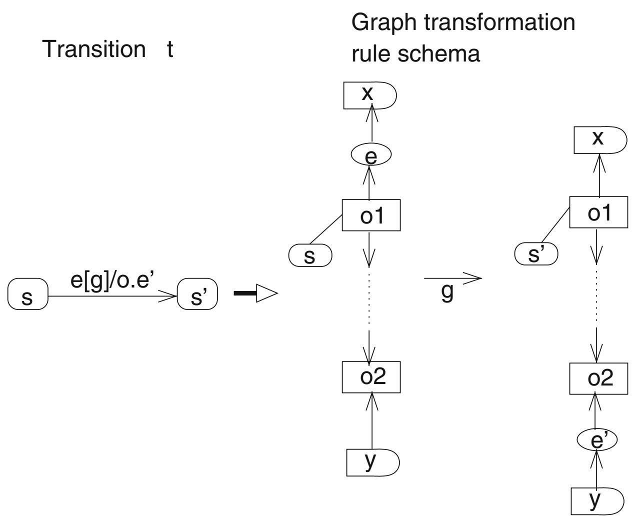


Fig. 7 The state changing rule schema

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_1, the change of the state of o_1 from s to s', and the insertion of e' in the event queue of some object, say o_2, of the class to which the path o leads. For this purpose the rule contains in its left- and right-hand side the object o_1 (i.e., more precisely a node labeled with a variable of type c standing for any object of type c), the object o_2 and the path from o_1 to o_2 corresponding to the path o. This path is obtained from o by converting every association e in o with $m(e) = (a, r_1, r_2, x_1, x_2)$ into a link with label (a, r_1, r_2) and every class c into an object labeled with a variable of class c. The state s is associated with o_1 in the left-hand side of the rule and changed to the state s' in the right-hand side. The event e is connected to o_1 in the left-hand side whereas in the right-hand side e' is connected to the object o_2.

The construction of the graph transformation rule for t can be done automatically as indicated in Fig. 7. On the left-hand side of Fig. 7, the transition t is depicted. The corresponding graph transformation rule schema is shown on the right of the figure where objects are denoted by rectangles, states by rectangles with rounded corners, and events by ellipses. The arrows $\longrightarrow \cdots \longrightarrow$ from o_1 to o_2 constitute an instantiation of the path o. The guard g of the rule must be checked before its application. This is indicated by denoting g below the arrow pointing from the left-hand side to the right-hand side of the rule. The application of the rule changes the state s of o_1 to s', deletes event e from the event queue of o_1 if it is the first event in the queue, and inserts e' at the end of the event queue of o_2.

In the host graphs such a rule is applied to, every object points to the first event of its event queue which in turn points to the next and so on. The last event in the queue points back to the object. If the event queue is empty, it is represented as a loop. The graph transformation rules do not contain the entire event queues. They include only the beginning of the queue of o_1 and the end of the queue of o_2. When applying such a rule, the first event e in the event queue of o_1 is deleted so that o_1 will then point to x, which is either another event or o_1 itself if the removed event was the only event in the queue. This means that in the application of the rule the x node can be mapped to the second event in the event queue of o_1 or to o_1 itself if there is no second event. Moreover, the event queue of o_2 in the host graph can be empty or not. In the first case, the y-labeled node is mapped to o_2 whereas in the second case it is mapped to the last event in the event queue of o_2. That is why we depict the x and the y node as a mixture of ellipse and rectangle. Hence, the type of the variables x and y is the union of the type containing all events and the type containing all objects.

The rule for the transition from the state *HasMailed* to *HasTyped* in the state diagram of the class *Secretary* is depicted in Fig. 8. It contains objects of class *Secretary* and *Printer*. On the left-hand side the *Secretary* object is attached to the state *HasMailed*. On the right-hand side, the state *HasTyped* is attached to the *Secretary*. The *Secretary* on the left-hand side has a pointer to the first event *type*(t) of its event queue. Applying the rule this event is deleted from the event queue of the *Secretary*, and the event *print*(t.*letter*) is inserted at the end of the printer's event queue on the right-hand side.

Please note that for a correct implementation of our approach the parameters of the call events point to the objects they represent. This additional technical information can be added to the rules in a straight-forward way and is omitted here for a better readability of the rules. In our example we will identify the parameter object by giving to it the same name as to the parameter in the event.

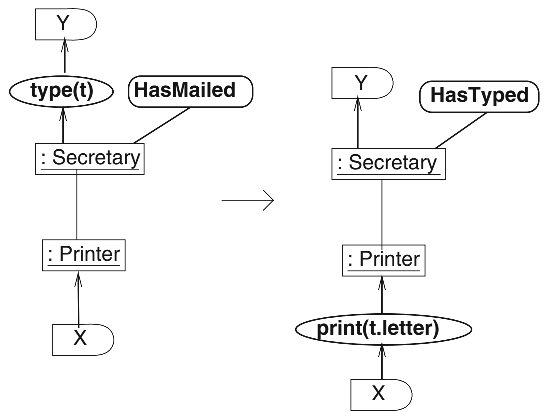


Fig. 8 The rule for the transition *type* of Fig. 4

4 Integration of class, object, and state diagrams

Class and state diagrams can be integrated in such a way that every class is connected with the state diagram describing its behavior. This leads to the notion of integrated diagrams. In an integrated specification, integrated diagrams are transformed via graph transformation rules that are obtained based on the combination of the graph transformation rules associated with the class diagram and the graph transformation rules associated with the state diagrams.

4.1 Integrated diagrams

An *integrated diagram* is a pair INTD = (CD, mstd) where CD is a class diagram and mstd: $V_{\mathrm{CD}} \to \mathrm{SD}$ is a mapping assigning a state diagram mstd(c) to every class c in CD such that mstd(c) contains only events of class c.

A *system state* of an integrated diagram (CD, mstd) is an object diagram that fits CD and where additionally every object is connected with a state of the state diagram associated with the class of the object. Moreover, as mentioned before, every object has an event queue that may be empty. It is worth noting that loops representing empty queues can be distinguished from self-links by labeling all edges pointing from or to an event with a special symbol, say queue. For reasons of a better readability, this is omitted here.

An example of a system state of the integrated diagram composed of the above class diagram and state diagrams is presented in Fig. 9.

The set of all system states can be formally specified in a rule-based way as follows:

- The initial graph can be any object diagram OD fitting CD, i.e., for which there exist mappings $g_V : V_{\mathrm{OD}} \to V_{\mathrm{CD}}$ and $g_E : E_{\mathrm{OD}} \to E_{\mathrm{CD}}$ as described in Sect. 2.
- There is a (parameterized) rule that adds exactly one state state(o) and one empty event queue queue(o) to every object o in OD such that state(o) is contained in the state diagram associated with $g_V(o)$, i.e., state$(o) \in S_{\mathrm{mstd}(g_V(o))}$. The left-hand side of the rule consists of a node v with a variable x as label and is equal to the common part, the right-hand side consists of the node v, a state s, an edge going from v to s, and a loop from v to v labeled with queue. The requirements that there must be added exactly one state and one event queue to every object, and that the state s must belong to the class of the node to which v is mapped when applying the rule, can be realized by appropriate application conditions.
- To every object, a sequence of events is added. The left-hand side of the corresponding rule consists of an object node v and a node v' that can be mapped to an event or an object, and a queue-labeled edge from v' to v. The common part consists of v and v', and the right-hand side consists of v, v', a new node v'' labeled with an event occurring in the state diagram of the object to which v is mapped, plus two queue-labeled edges e and e' where e points from v'' to v and e' from v' to v''. Hence, this rule inserts an event at the end of an event queue.

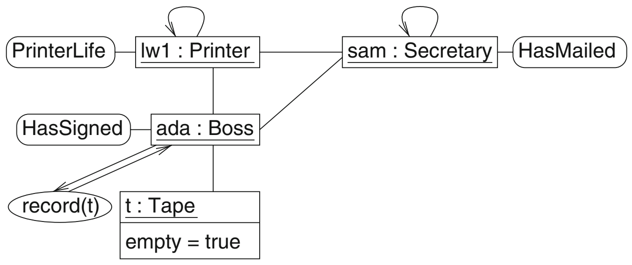


Fig. 9 An instance of an integrated diagram

4.2 Combining the rules of class and state diagrams

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. The transition rules of state diagrams are glued with the rules of the classes they are associated with by identifying common objects. More precisely, let $r = (L, K, R)$ be a graph transformation rule modeling an operation op of class c, let $t = (s, e, g, o.e', s')$ be a transition of the state diagram associated with c such that the event e is equal to op, and let $r' = (L', K', R')$ be the rule constructed for t as described in Sect. 3.3 and depicted in Fig. 7. Then the integrated rule $(int(L), int(K), int(R))$ is automatically obtained according to the following steps.

1. Construct the *interface rule* ir = (IL, IK, IR) of r and r', where IL = IK = IR is the graph consisting of the node mainobject(r), i.e., the object node that represents the object that executes the operation op. Then, obviously, IL $\subseteq L$, IK $\subseteq K$, and IR $\subseteq R$. Moreover, let gl$'$: IL $\to L'$, gk$'$: IK $\to K'$, and gr$'$: IR $\to R'$ be defined such that mainobject(r) is mapped to the node $o1$ in Fig. 7, i.e., the node that represents the object has fires the transition.
2. Construct a new *integrated rule* by gluing r and r' in their common part ir. This can be done by first unifying r and r' disjointly and then identifying all items that correspond to the same element in ir. Formally, this gluing of graphs can be obtained via the pushout constructions of gl and gl$'$, gk and gk$'$, and gr and gr$'$ where gl, gk, and gr are inclusions (see [3] for more details), but it can also be described in the set-theoretic way of Sect. 3.

For example, for the transition rule *type* in Fig. 8 and the rule for *Secretary::type(t)* in Fig. 5 the interface rule consists of a secretary object in its left- and its right-hand side which

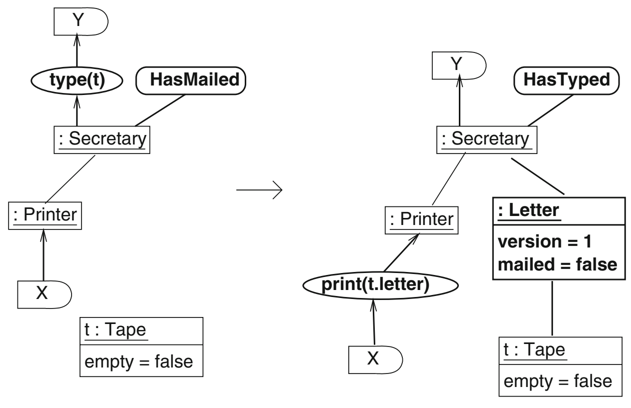


Fig. 10 The integrated rule for *type*

are mapped to the secretary nodes of the rules in Figs. 5 and 8. The integrated rule is depicted in Fig. 10 and is obtained by gluing both rules in their secretary objects. It models the typing of a letter provided that the secretary is in the state *HasMailed* and has the event *type*(t) at the top of the event queue.

Figures 11, 12 and 13 depict further integrated rules for our running example. The set of all integrated graph transformation rules which can be associated in the described way

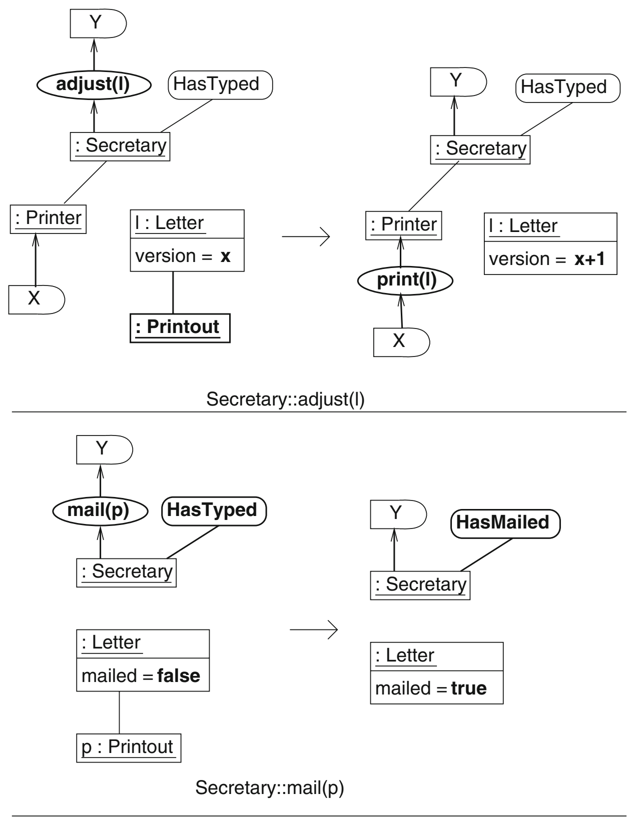


Fig. 11 Further integrated rules for *Secretary* operations

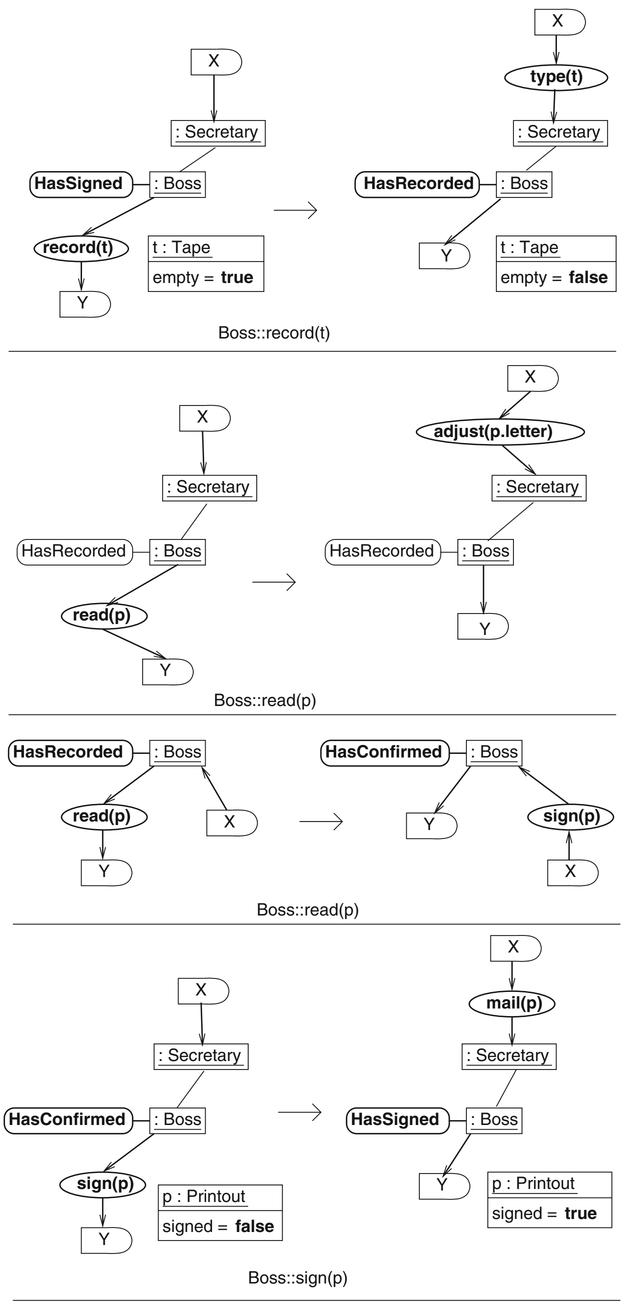


Fig. 12 Integrated rules for *Boss* operations

with an integrated diagram INTD is called the *set of integrated rules for* INTD.

4.3 Integrated specifications and their semantics

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 the set of integrated rules for INTD. In the system state I, all objects are in their initial states, all event queues up to one are empty, and

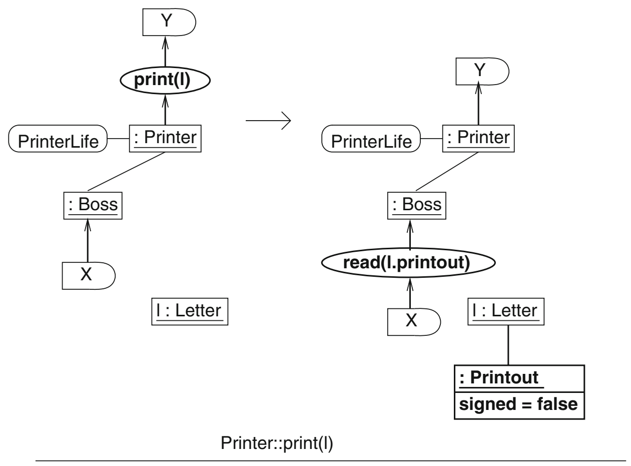


Fig. 13 Integrated rule for *Printer* operation

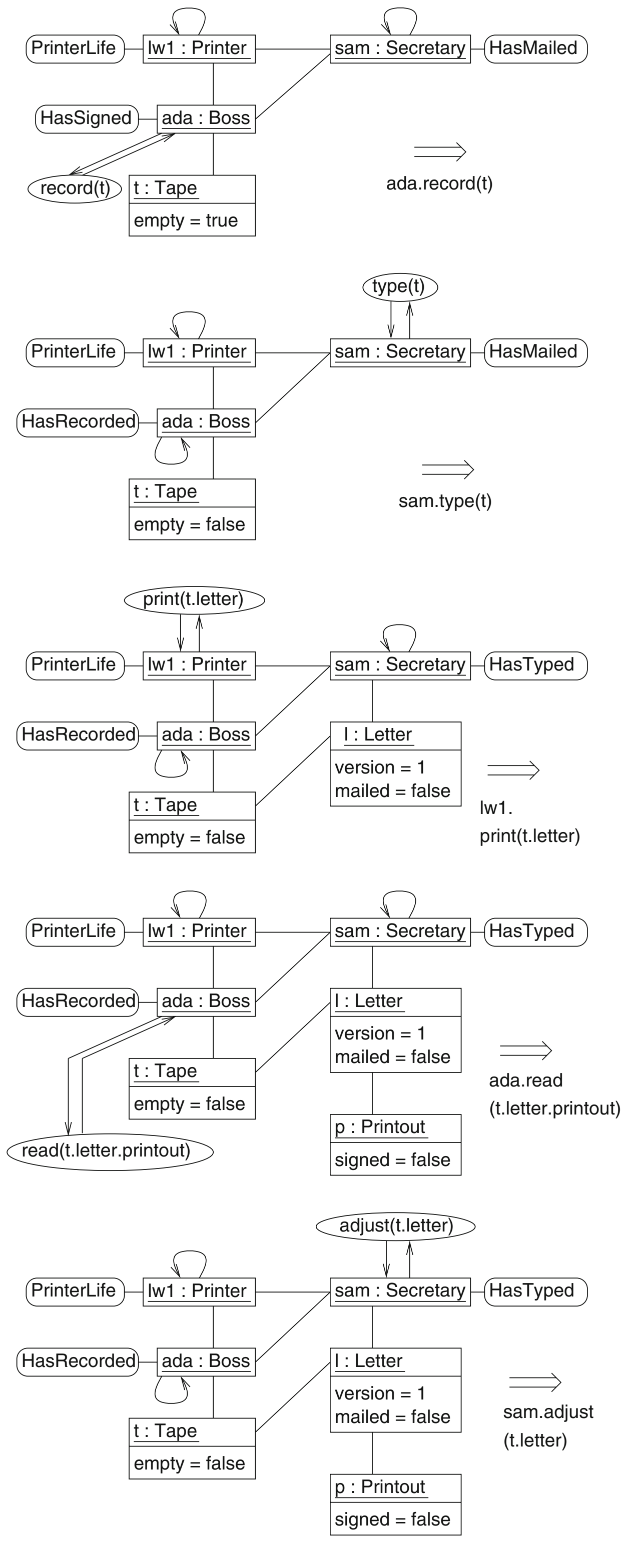


Fig. 14 Derivation (part 1)

the only non-empty event queue contains the event of a transition, the source of which is an initial state. The *semantics* of an integrated specification is denoted by SEM(INTD, I, R) and consists of all the derivations $G \overset{*}{\underset{R}{\Longrightarrow}} G'$ such that $G \cong I$.

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. The initial diagram is the integrated diagram of Fig. 9 and R consists of the rules presented in Fig. 11, 12 and 13. Figures 14 and 15 illustrate how the different system states (i.e., system states represented by integrated diagrams) can be derived with the example specification. The derivation starts with the integrated diagram of Fig. 9 and applies at first the rule *ada.record*(t). This means that after dispatching the event *record*(t) the attribute *empty* of the tape is changed from *true* to *false*. Additionally, the event *record*(t) is deleted from the event queue of *ada*, *type*(t) is inserted in the event queue of *sam*, and the state of *ada* changes from *HasSigned* to *HasRecorded*. Afterwards *sam.type*(t) is applied which changes the state of *sam* to *HasTyped*, deletes *type*(t) from its event queue, and inserts *print*($t.letter$) in the event queue of *ada*. Moreover, a letter l is created and linked to tape t and *sam*. The rest of the derivation models the following process: letter l is printed and then read by *ada*. Afterwards it is adjusted by *sam*, printed again, and read again by *ada*. Finally, the printout is signed by *ada* and mailed by *sam*.

5 Integrating sequence and collaboration diagrams

State diagrams describe the behavior of individual objects. It is very difficult to understand the interactions of different objects only by looking at the set of state diagrams. For this purpose, UML offers interaction diagrams, i.e., sequence and collaboration diagrams.

5.1 Sequence and collaboration diagrams

We only consider interaction diagrams at instance level, which consist of objects sending messages to each other.

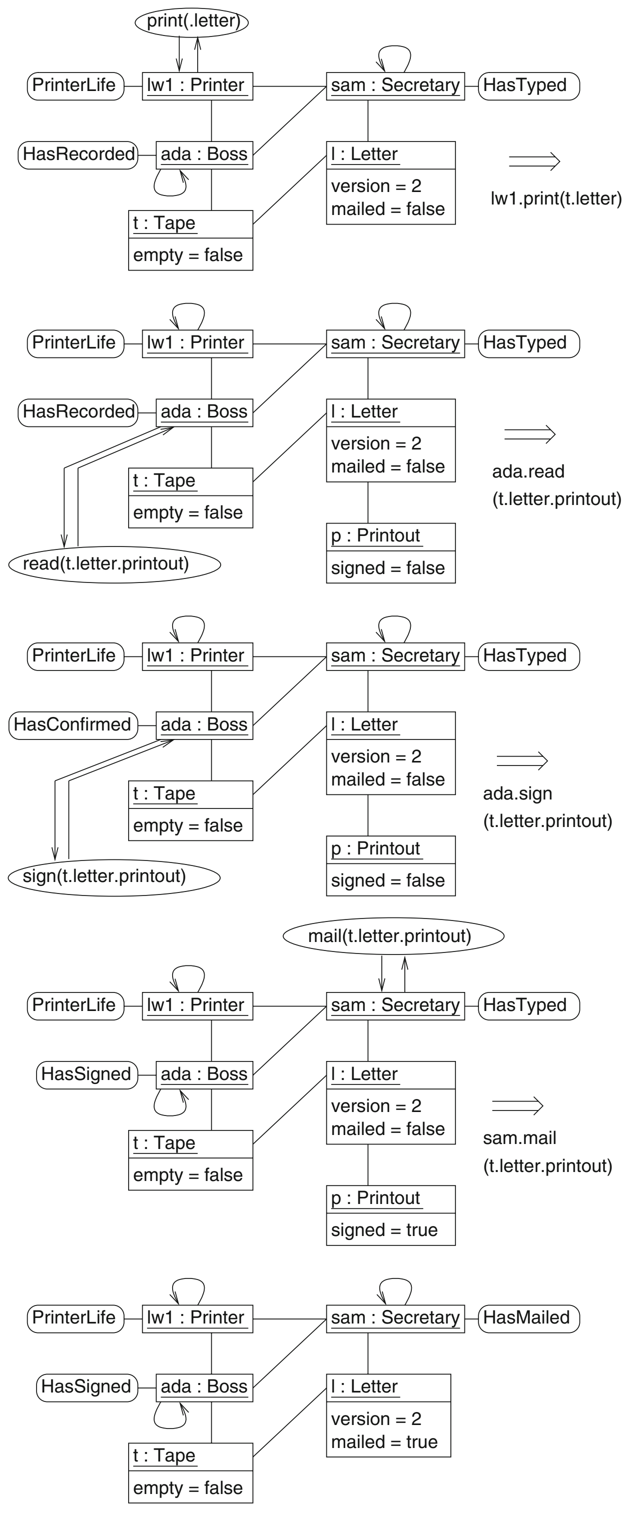


Fig. 15 Derivation (part 2)

Such a diagram represents a part of a concrete system execution. Sequence and collaboration diagrams contain basically the same information, but focus on different aspects, which are discussed in the following paragraphs (see also [7]).

Sequence diagrams display interactions in two dimensions. The horizontal dimension shows objects while the vertical dimension represents time. A vertical lifeline is connected to each object. Messages are shown as labeled arrows from the lifeline of the sending object to the lifeline of the receiving object. The arrows are ordered along the vertical time axis, i.e., those closer to the top are sent earlier than those further below.

A collaboration diagram (at instance level) is an object diagram with superimposed behavior. Numbered messages can be attached to the links, together with an arrow indicating the direction. There are some other features available in collaboration diagrams we do not consider here. In their basic form, collaboration and sequence diagrams offer different views on the same information: the sequence diagram emphasizes *time* aspects by a message ordering from top to bottom, whereas the collaboration diagram emphasizes *structure* aspects by explicitly showing the links between the objects (and expressing the message sequence only by a numbering system).

5.2 Relating sequence and collaboration diagrams to derivations

Every derivation in the graph transformation system can be mapped to a sequence and a collaboration diagram. Fig. 16 shows the sequence diagram corresponding to the derivation in Figs. 14 and 15. It contains a *Boss*, a *Secretary* and a *Printer* object. The first rule application *ada.record*() removes the event *record*(t) from the queue of the *Boss* object and puts the event *type*(t) in the queue of the *Secretary* object, i,e., during the first rule application the boss sends the message $type(t)$ to her secretary. This rule application corresponds to the first arrow in the sequence diagram from *Boss* to *Secretary*, labeled with *type*(t). (Please note that the application of the rule *record*(t)corresponds to an arrow labeled with *type*(t), because rules are labeled with events whereas arrows are labeled with call actions.) The last but one rule application $ada.sign(t.letter.printout)$ corresponds to the arrow from *Boss* to *Secretary*, labeled with $mail(t.letter.printout)$. The last rule application $sam.mail(t.letter.printout)$ is not mapped into the sequence diagram, because it does not put an event in any queue. The first graph of the derivation contains already the event *record*(t) in the queue of the *Boss* object. Nothing is said about how and when it was put there, so there is no arrow labeled with *record*(t) in the sequence diagram.

Due to the fact that the arrows in sequence diagrams are labeled with call actions and the rules with call events, rule applications that do not insert a call event into the event queue of some object are not reflected in interaction diagrams. Hence, different derivations can be mapped to the same collaboration/sequence diagram. For example, the derivation in Figs. 14 and 15 excluding the last rule application also maps to the sequence diagram shown in Fig. 16.

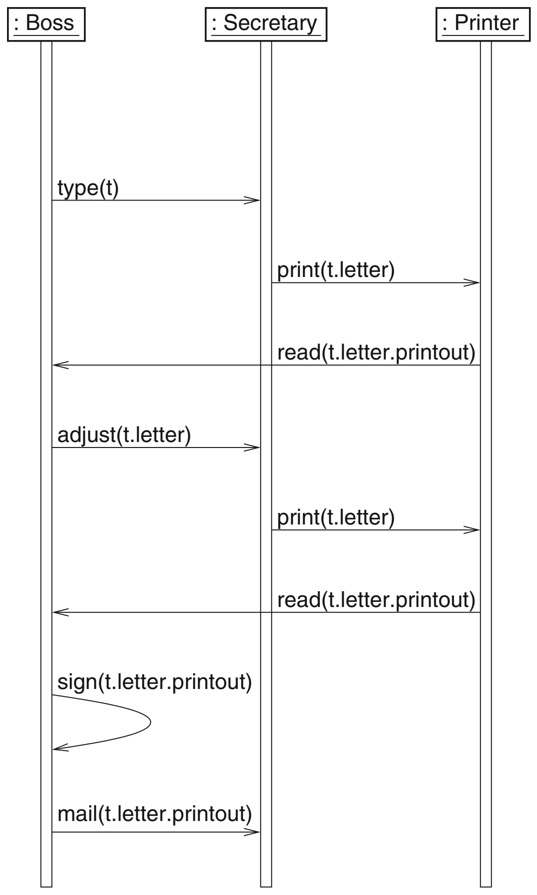


Fig. 16 Sequence diagram for the derivations in Figs. 14 and 15

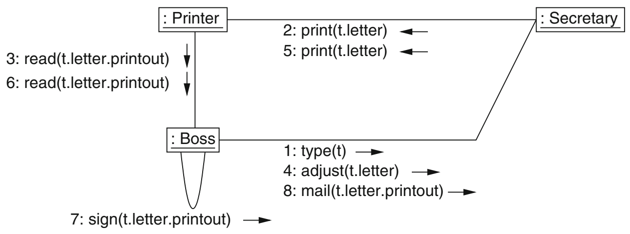

: Printer
: Secretary
2: print(t.letter)
5: print(t.letter)
3: read(t.letter.printout)
6: read(t.letter.printout)
: Boss
1: type(t)
4: adjust(t.letter)
8: mail(t.letter.printout)
7: sign(t.letter.printout)

Fig. 17 Collaboration diagram for the derivations in Figs. 14 and 15

To construct a collaboration diagram for a given derivation, we proceed as follows: (1) Every object that exists in the graph during the derivation is added to the collaboration diagram. (2) The creation of an event by a rule application corresponds to the sending of a message. In the order of the single derivation steps we add the messages to the collaboration diagram: A derivation step that removes an event from the queue of an object a and puts event e in the queue of object b leads to a message from a to b that calls the operation e. Hence, a link labeled with the call event e and a small arrow that indicates the direction of the message is inserted between a and b. (3) All objects that are not source or target of a message are removed. If there are no events created in the derivation, we would get an empty collaboration diagram. Fig. 17 depicts the collaboration diagram for the derivation in Figs. 14 and 15.

Given a set R of rules, we can associate with every interaction diagram ID a set ruleseq(ID) $\subseteq R^*$ of rule sequences as follows. In the order of the messages we choose rules to be applied: For a message e from an object a to an object b we have to find a rule that removes an event from the queue of a and put the event e into the queue of b. The semantics of the interaction diagram ID, denoted by SEM(ID), consists of all derivations $G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ such that $r_1 \cdots r_n \in$ ruleseq(ID).

To sum up, every derivation can be mapped to one sequence and collaboration diagram, and every valid sequence and collaboration diagram can be mapped to a non-empty set D of derivations such that every derivation in D reflects the sequence of message passing. But clearly, a derivation contains much more information, e.g., the effect of an operation call to objects, attributes and links.

5.3 Integrated specifications including interaction diagrams

Now we can redefine the concept of an integrated specification by including interaction diagrams. This leads to the definition IS = (INTD, I, R, ID) where (INTD, I, R) is defined as before and ID is an interaction diagram. The *semantics* of an integrated specification IS = (INTD, I, R, ID) consists of all derivations in SEM(INTD, I, R) $\cap$ SEM(ID).

Two interesting questions are (1) whether a derivation can be found for a given interaction diagram at all, that is whether the interaction diagram is valid, and (2) whether the interaction can occur in a given system state. The thorough answer of question (1) will be of future work. However, a very first approach towards a solution to this problem is to construct all derivations that only involve instantiations of the graphs in the rules (these are finitely many up to naming of objects) and to check whether one gets in this way a derivation the initial graph of which is or can be extended to a valid system state. This extension must be done so that no dangling edges can occur during the derivation. To illustrate this, we check whether there is a derivation for the sequence of integrated rules $Boss::record(t)$ $Secretary::type(t)$ that are the first two rules determined by the diagrams in Figs. 16 and 17. To this aim we proceed as follows. An instantiation of the left-hand side of $Secretary::type(t)$ and an instantiation of the right-hand side of $Boss::record(t)$ are glued together such that on one hand the rule $Secretary::type(t)$ can be applied to the resulting graph, say G, and on the other hand the rule $Boss::record(t)$ can be applied backwards to G. In the worst case, we have to consider all gluings in order to get an initial valid system state. One gluing is depicted in Fig. 18 and is obtained from amalgamating all common

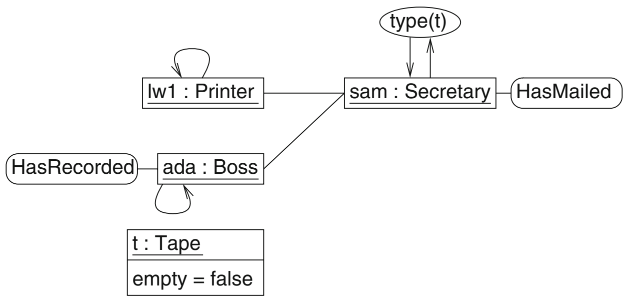


Fig. 18 Gluing of two rule sides

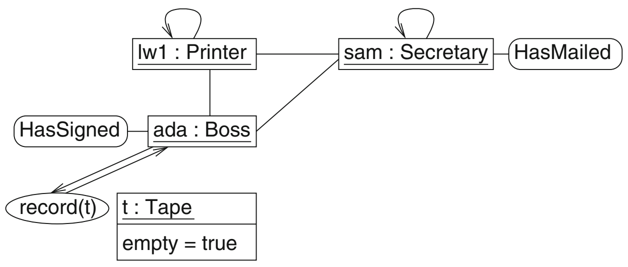


Fig. 19 Result of a reverse rule application

objects and links of the right-hand side of *Boss*::*record*(t) and the left-hand side of *Secretary*::*type*(t). (Please note that the variable Y of the left-hand side of *Secretary*::*type*(t) as well as the variable X of the right-hand side of *Boss*::*record*(t) are instantiated with the secretary object. The variable X of *Secretary*::*type*(t) is instantiated with the printer and the variable Y of *Boss*::*record*(t) is instantiated with the boss. The *Secretary* node is instantiated with *sam*, the *Boss* with *ada* and so on.)

Another way to glue these two instantiated graphs would be the disjoint union of both.

The reverse application of the rule *Boss*::*record*(t) to the graph of Fig. 18 results in the diagram depicted in Fig. 19 which can be easily extended to the system state in Fig. 9.

The generalization of this illustration towards an algorithmic solution to the case of arbitrary long sequences of rules is a topic of future research.

To answer the second question the modeler can check by example whether the specified interaction can occur in states where it should and cannot occur in states where it should not. Thus the formalization of UML diagrams by graph transformation gives feedback to the modeler about the applicability of the message sequence specified in the interaction diagram.

In the system state depicted in Fig. 20, the sequence of messages modeled in Figs. 16 and 17 is not applicable for two reasons. The only event the diagram shows is *record*(t), so the rule for *record*(t) is the only one that should be considered for application. However, the attribute *empty* of tape t has not the value $true$. But even if it had, the sequence would not be applicable since the secretary is in state *HasTyped*, in which

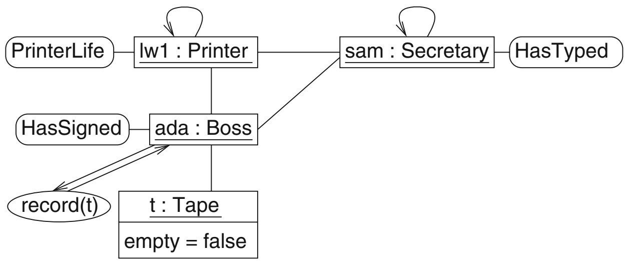


Fig. 20 System state not being an initial state for the interactions specified in the diagrams of Fig. 16 and 17

she does not react to *type* events. This is reasonable and is due to the following correctness properties of the integrated specification: No boss can record something on a full tape and no secretary can type a letter if he is in the state *HasTyped*.

With the first graph in Fig. 14 as initial graph, a mapping can be found from objects in the graph to objects in the sequence diagram, and a rule application for each arrow can also be found. If the modeler rates this sequence of system states as reasonable, this would reinforce the modeler's belief in the correctness of the model. Otherwise, either the interaction diagrams or the model, i.e., the integrated specification has to be changed.

6 Related work

Much research has been done concerning the formalization of UML semantics, so that it goes beyond the scope of this paper to refer to all of them. Hence, in this section we mention only a selection of contributions to the formalization of UML semantics. First of all there exist many papers that study the formalization of state diagrams. This seems to be natural since state diagrams specify the dynamic behavior of objects. Hence, the first of the following paragraphs mentions different approaches to formalize the operational semantics of state diagrams. The second paragraph gives a slight insight into other approaches that deal with an integrated UML semantics. Since we propose graph transformation as formal model, the last paragraph contains a selection of further papers that bring UML diagrams and the theory of graph transformation together.

Operational semantics of state diagrams. In the literature there exist a series of approaches that formalize the operational semantics of state diagrams. They mainly differ in the underlying formal methods. More precisely, Schettini and Peron [32,33], Kuske [27], and Varró [46] define configurations of state diagrams as graphs so that every run-to-completion step includes the application of one or a set of graph transformation rules. Clearly, these approaches follow the same basic ideas as we do, but they are concentrated on a single diagram type. Another approach is presented by Lilius and Paltor in [29,30] where state diagrams are translated into (conditional) term rewriting systems and then into input languages for specific model-checkers. In [17], Gnesi, Latella and Massink represent state diagrams as hierarchical automata the operational semantics of which is given by labeled transition systems, that in turn can be used as a model for proving the satisfiability of logic formulas. Labeled transition systems are also used as the formal basis for the semantics of state diagrams by Reggio et al. [35] and von der Beek [2]. In [39], Rossi, Enciso, and Guzmán give a state diagram semantics based on temporal logic. A compositional semantics of state diagrams based on set theory is presented by Simons in [43]. Another approach that translates state diagrams and collaboration diagrams into Petri nets is given by Hu and Shatz in [25]. Although some of the mentioned papers use additionally interaction diagrams to describe the behaviour of several state diagrams or to represent counter-examples, they do not focus on the formal and explicit integration of different diagram types into one the same formal framework.

Integrated semantics. In [1], Baresi and Pezzè study how class, state, and collaboration diagrams can be automatically translated into high-level Petri nets via the so called CR-approach which is based on graph transformation. Moreover, it is discussed how required properties of UML specifications could be verified on the formal model. The translation of a UML specification into a formal model via the CR-approach needs three types of rules, one for translating the UML syntax, another one for translating the UML semantics, and a third one that allows to visualize situations in the formal model in a UML-like manner. As we have illustrated in this paper, such a translation into a formal model and back again is not necessary if one takes graph transformation as the formal model, because in this case the only requirements are that diagrams be regarded as graphs and class operations as graph transformation rules. In [49,50], Ziemann, Hölscher, and Gogolla introduce a similar approach of an integrated semantics of UML that mainly differs from the one presented in this paper in the following aspects: In [49,50] class operations are specified with collaboration diagams which contain a set of names of suboperations associated with an order prescribing their application order. These collaboration diagrams are translated into sets of graph transformation rules that are applied in the specified order. Many of these graph transformation rules model basic operations like the creation/deletion of an object or a link, or the setting of an attribute. In contrast, in our approach a class operation is modeled by a single graph transformation rule which performs a series of such basic operations in one application step so that the modeler can directly specify the visual effect of an operation call. For example, the setting of an attribute is modeled in [49,50] with two graph transformation rules, whereas in our approach an attribute is set within the application of one graph transformation rule which can additionally have further effects like the creation of a series of new links, etc. Moreover, in [49,50], system states are represented in a very complex way so that it is difficult to understand what they represent, i.e., the benefits of a visual representation get mostly lost. For example, the insertion of a link results in a graph transformation rule the left-hand side of which consists of eight nodes and six edges and the right-hand side of eleven nodes and thirteen edges. In our approach the insertion of a link is modeled with a much simpler rule consisting of two nodes in its left and right-hand side and an additional edge in its right-hand side. On the other hand, [49,50] integrates also use case diagrams which are not considered in this approach but we are quite sure that they can be integrated straightforwardly in an analogous way. Hence, the approach in [49,50] is somewhat nearer to UML because it models class operations by collaboration diagrams and integrates use case diagrams. On the other hand it is harder to understand because system states and the rule sets modeling class operations do not have an intuitive visualization due to the fact that they represent a lot of technical details.

UML diagrams and graph transformation. Apart from the already mentioned papers that use graph transformation for formalizing state diagrams, there remain other papers that relate UML diagrams with graph transformation that are worth to be mentioned. In [13], Engels et al. transform collaboration diagrams into graph transformation rules with the aim to provide an interpreter and to allow modeling at the meta-model level. Varró and Pataricza [47] propose a graph transformation-based framework for defining the semantics of mathematical models in a UML notation. Bottoni, Parisi-Presicce, and Taentzer [5] present a graph transformation approach to maintain code and UML specifications consistent. Cordes, Hölscher, and Kreowski [7] present a translation of sequence diagrams into collaboration diagrams that is based on graph transformation rules. In [22], Hausmann, Heckel, and Taentzer propose a formal interpretation of UML use case, activity, and collaboration diagrams based on concepts from the theory of graph transformation. In [15], Engels, Heckel, and Küster present meta-model based mapping rules that translate elements of UML models into a semantic domain. Those rules consist of a meta-model part M, a part of a concrete UML diagram D and the component of the semantic domain to which D should be translated. In [24], Heckel, Küster and Taentzer propose triple grammars and attributed graph transformation for defining such meta-model mapping rules. In [14], Engels et al. propose dynamic meta-modeling rules as a notation for describing consistency conditions for UML diagrams. In [42], Schmidt and Varró present the tool CheckVML that can be used for checking dynamic consistency properties of UML models. In [16], Engels, Heckel, and Küster introduce the Consistency Workbench, which is a tool for defining and establishing consistency in a UML-based development process. Both tools are based on graph transformation.

In many cases, the basic idea coincides with ours, i.e., to describe the meaning of UML diagrams by means of transformation rules on suitable graphical or similar structures. But while most other approaches focus on one diagram type or combine only a few of them, our intention is to integrate many or even all types of diagrams into one semantical framework that not only provides the diagrams with meaning, but also covers their interaction.

7 Conclusion

We have introduced a graph transformational description for central language features of UML. In our approach, system states are represented as object diagrams combined with object states and event queues. Operations from class diagrams and transitions from state diagrams are described by single graph transformation rules, respectively. These rules are combined into integrated rules that manipulate system states. Every application of an integrated rule models the firing of a transition, i.e., in every transformation step, the event of the transition is executed and the event queues as well as the current states of the involved objects are updated. The integrated rules together with an initial system state yield a coherent single graph transformation system representing the integrated semantics of the class, object, and state diagrams of an UML model.

Moreover, we have shown how interaction diagrams can be integrated into this approach. These diagrams specify interactions of objects, i.e., sequences of messages sent from one object to another. A message requests an operation execution and therefore corresponds to the creation of a call event. Since most rules not only consume but also create events, there is a close relationship between interaction diagrams and derivations of the graph transformation system. An interaction diagram can be found for every derivation. On the other hand, we sketched how it can be checked whether there exists a derivation for an interaction diagram. In this case the interaction diagram is consistent with the system modeled by class and state diagrams and formalized by the graph transformation system.

Our approach provides various benefits:

1. *Syntax check*: UML diagrams with incorrect syntax do not have a formalization in form of a graph transformation system.
2. *Validation*: The graph transformation system can be used to validate that the described system meets the intended system by (1) applying rules to system state graphs and examining the resulting graphs and (2) checking whether an interaction modeled in a sequence or collaboration diagram can occur in a system state in which it should and cannot occur in system states in which it should not.
3. *Verification*: Properties of states and state transitions can be verified. Referring to our running example, it can be verified that, e.g., (1) printouts never change from signed to unsigned, (2) secretaries do not mail unsigned printouts and (3) version numbers of letters are never negative. Those properties directly follow from the absence of transformation rules with the respective effects. Nevertheless, as soon as model-checkers for graph transformation systems become available verification of UML specifications based on graph transformation can be automated. In this context it is worth noting that a model-checker for graph transformation systems is being developed within the *GROOVE* project (where the name *GROOVE* stands for GRaphs for Object-Oriented VErification) at the University of Twente (cf. [26,36,37]).

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

1. The presented integrated semantics covers only basic features of UML. Hence it has to be investigated how other language elements like composite states, different kinds of events or asynchronous messages can be handled.
2. 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 that allow to encapsulate sets of graph transformation rules and which provide control mechanisms for the application process of rules (cf. [23]).
3. In complex cases the integration of various UML diagrams may lead to large diagrams which are difficult to handle and to understand. Therefore, for practical use, structuring concepts for graphs should be incorporated in the presented approach (cf., e.g, [6,44]).
4. To be able to use our approach in practice, adequate transformation tools are needed. It should be thoroughly investigated in which way existing tools can be employed to achieve this aim. Just to mention a few, we believe that for example the *AGG*-system [45] from the Technical University of Berlin could be used for specification simulation and the *GROOVE*-system for verification.
5. How do we cope with under-specification? It is desirable that a UML model can be translated into a graph transformation system even if important information is missing, such as semantics of operations in classes.

6. So far, the approach requires that the semantics of operations be given as graph transformation rules with object diagrams as graphs. In UML, these rules can be represented as two object diagrams with a ⟨⟨become⟩⟩ flow relationship in between. Nevertheless, it should also be examined if operations could be specified in a suitable way with interaction diagrams.

Acknowledgments We are grateful to the anonymous reviewers for their valuable comments on a previous version of this paper.

References

1. Baresi, L., Pezzè, M.: On formalizing UML with high-level Petri nets. In: Agha, G., Cindio, F.D. (eds.) Proceedings of Concurrent Object-Oriented Programming and Petri Nets. Lecture Notes in Computer Science, vol. 2001, pp. 271–300. Springer, Heidelberg (2001)
2. von der Beek, M.: A structured operational semantics for UML-statecharts. Softw. Syst. Model. **1**(2), 130–141 (2002)
3. Boehm P., Fonio H.R., Habel A. Amalgamation of graph transformations: a synchronization mechanism. J. Comput. Syst. Sci. 34:377–408 (1987)
4. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1998)
5. Bottoni, P., Parisi-Presicce, F., Taentzer, G.: Coordinated distributed diagram transformation for software evolution. In: Heckel, R., Mens, T., Wermelinger, M. (eds) Proceedings of the Workshop on 'Software Evolution Through Transformations' (SET'02), Electronic Notes in Theoretical Computer Science, vol. 72 (2002)
6. Busatto, G., Kreowski, H.J., Kuske, S.: Abstract hierarchical graph transformation. Math. Struct. Comput. Sci. **15**(04), 773–819 (2005)
7. Cordes, B., Hölscher, K., Kreowski, H.J.: UML interaction diagrams: correct translation of sequence diagrams into collaboration diagrams. In: Nagl, M., Pfalz, J. (eds.) Applications of Graph Transformations with Industrial Relevance (AGTIVE), no. 3062 in Lecture Notes in Computer Science, pp. 275–291. Springer, Heidelberg (2003)
8. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In: Rozenberg [40], pp. 163–245
9. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools. World Scientific, Singapore (1999)
10. Ehrig, H., Habel, A.: Graph grammars with application conditions. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L, pp. 87–100. Springer, Berlin (1986)
11. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution. World Scientific, Singapore (1999)
12. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Parisi-Presicce, F., Bottoni, P., Engles, G. (eds.) Proceedings of 2nd International Conference on Graph Transformation (ICGT'04), Lecture Notes in Computer Science, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
13. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds) Proceedings of UML 2000—The Unified Modeling Language. Advancing the Standard, Lecture Notes in Computer Science, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
14. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Testing the consistency of dynamic UML diagrams. In: Proceedings of 6th International Conference on Integrated Design and Process Technology (IDPT 2002), 23–28 June 2002, Pasadena (2002)
15. Engels, G., Heckel, R., Küster, J.: Rule-based specification of behavioral consistency based on the UML meta-model. In: Gogolla, M., Kobryn, C. (eds.) UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Lecture Notes in Computer Science, vol. 2185, pp. 272–286. Springer, Heidelberg (2001)
16. Engels, G., Heckel, R., Küster, J.M.: The consistency workbench: A tool for consistency management in UML-based development. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003—The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA. Proceedings, Lecture Notes in Computer Science, vol. 2863, pp. 356–359. Springer, Heidelberg (2003)
17. Gnesi, S., Latella, D., Massink, M.: Modular semantics for a UML state diagrams kernel and their execution to multicharts and branching time model-checking. J. Logic Algebr. Program. **51**(1), 43–75 (2002)
18. Gogolla, M., Richters, M.: Transformation rules for UML class diagrams. In: Bézivin, J., Muller, P.A. (eds.) The Unified Modeling Language, UML'98 - Beyond the Notation. Ist International Workshop, Mulhouse, France, June 1998, Selected Papers, LNCS, vol. 1618, pp. 92–106. Springer, Heidelberg (1999)
19. Gogolla, M., Richters, M.: Expressing UML class diagrams properties with OCL. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL, The Rationale behind the Object Constraint Language, Lecture Notes in Computer Science, vol. 2263, pp. 86–115. Springer, Heidelberg (2002)
20. Gogolla, M., Ziemann, P., Kuske, S.: Towards an integrated graph based semantics for UML. In: Graph Transformation and Visual Modeling Techniques (GT-VMT 2002), Electronic Notes in Theoretical Computer Science, vol. 72 (2003)
21. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26**(3,4), 287–313 (1996)
22. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach. In: Proceedings of the 24th International Conference on Software Engineering 2002, Orlando, USA, pp. 105–115. IEEE Computer Society Press (2002)
23. Heckel, R., Engels, G., Ehrig, H., Taentzer, G.: Classification and comparison of module concepts for graph transformation systems. In: Ehrig et al. [9], pp. 639–689
24. Heckel, R., Küster, J.M., Taentzer, G.: Towards automatic translation of UML models into semantic domains. In: Kreowski, H.J., Knirsch, P. (eds.) Applied Graph Transformation (AGT'02), pp. 11–22 (2002)
25. Hu, Z., Shatz, S.M.: Mapping UML diagrams to a petri net notation for system simulation. In: Proceedings of International Conference on Software Engineering and Knowledge Engineering, pp. 213–219 (2004)
26. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) Proceedings of 13th International Workshop on Software Model Checking (SPIN'06), no. 3925 in Lecture Notes in Computer Science, pp. 299–305. Springer, Heidelberg (2006)
27. Kuske, S.: A formal semantics of UML state machines based on structured graph transformation. In: Gogolla, M., Kobryn, C. (eds.) UML 2001—The Unified Modeling Language. Modeling languages, Concepts, and Tools, Lecture Notes in Computer Science, vol. 2185, pp. 241–256. Springer, Heidelberg (2001)

28. Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.J.: An integrated semantics for UML class, object, and state diagrams based on graph transformation. In: Butler, M., Sere, K. (eds.) 3rd International Conference on Integrated Formal Methods (IFM'02), Lecture Notes in Computer Science, vol. 2335, pp. 11–28. Springer, Heidelberg (2002)
29. Kwon, G.: Rewrite rules and operational semantics for model checking UML statecharts. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000 - The Unified Modeling Language. Advancing the Standard. 3rd International Conference, York, UK, October 2000, Proceedings. *Lecture Notes in Computer Science*, vol. 1939, pp. 528–540. Springer, Heidelberg (2000)
30. Lilius, J., Paltor, I.: Formalising UML state machines for model checking. In: France, R., Rumpe, B. (eds.) Proceedings of UML'99—The Unified Modeling Language. Beyond the Standard, Lecture Notes in Computer Science, vol. 1723, pp. 430–445. Springer, Heidelberg (1999)
31. Löwe, M., Korff, M., Wagner, A.: An algebraic framework for the transformation of attributed graphs. In: Sleep, M.R., Plasmeijer, R., van Eekelen, M. (eds.) Term Graph Rewriting, Theory and Practice, pp. 185–199. Wiley, Chichester (1993)
32. Maggiolo-Schettini, A., Peron, A.: Semantics of full statecharts based on graph rewriting. In: Schneider, H.J., Ehrig, H. (eds.) Proceedings of Graph Transformation in Computer Science, Lecture Notes in Computer Science, vol. 776, pp. 265–279. Springer, Heidelberg (1994)
33. Maggiolo-Schettini, A., Peron, A.: A Graph Rewriting Framework for Statecharts Semantics. In: Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G. (eds.) Proceedings of 5th International Workshop on Graph Grammars and their Application to Computer Science, vol. 1073, pp. 107–121. Springer, Heidelberg (1996). citeseer.nj.nec.com/article/maggiolo-schettini96graph.html
34. OMG: Unified Modeling Language specifcation, version 1.5 (2003). Available at http://www.omg.org/
35. Reggio, G., Astesiano, E., Choppy, C., Hussmann, H.: Analysing UML active classes and associated state machines—a lightweight formal approach. In: Maibaum, T. (ed.) Proceedings of Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany, Lecture Notes in Computer Science, vol. 1783, pp. 127–146. Springer, Heidelberg (2000)
36. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Nagl, M., Pfalz, J. (eds.) Applications of Graph Transformations with Industrial Relevance (AGTIVE), no. 3062 in Lecture Notes in Computer Science, pp. 479–485. Springer, Heidelberg (2003)
37. Rensink, A.: Towards model checking graph grammars. In: Leuschel, S.G.M., Presti, S.L. (eds.) Proceedings of 3rd Workshop on Automated Verification of Critical Systems, no. 3062 in Tech. Report DSSE-TR-2003-2, pp. 150–160. University of Southampton (2003)
38. Richters, M., Gogolla, M.: OCL: Syntax, semantics, and tools. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL: The Rationale behind the Object Constraint Language, pp. 42–68. Springer, Heidelberg (2002)
39. Rossi, C., Enciso, M., de Guzmán, I.P.: Formalization of UML state machines using temporal logic. Softw. Syst. Model. **3**, 31–54 (2004)
40. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations. World Scientific, Singapore (1997)
41. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading (1998)
42. Schmidt, Á., Varró, D.: CheckVML: A tool for model checking visual modeling languages. In: P. Stevens, J. Whittle, G. Booch (eds.) UML 2003—The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA. Proceedings, Lecture Notes in Computer Science, vol. 2863, pp. 92–95. Springer, Heidelberg (2003)
43. Simons, A.J.H.: On the compositional properties of UML statechart diagrams. Electronic Workshops in Computing: Rigorous Object-Oriented Methods, pp. 8/1–8/12 (2000)
44. Taentzer G. (1996) Hierarchically distributed graph transformation. In: Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G. (eds.) Proceedings of Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 1073, pp. 304–320 (1996)
45. Taentzer, G., Ermel, C., Rudolf, M.: The AGG-approach: Language and tool environment. In: Ehrig et al. [9], pp. 551–603
46. Varró, D.: A formal semantics of UML statecharts by model transition systems. In: Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G. (eds.) Graph Transformation. Ist International Conference, ICGT 2002, Barcelona, Spain. Proceedings, *Lecture Notes in Computer Science*, vol. 2505, pp. 378–392. Springer, Heidelberg (2002)
47. Varró, D., Pataricza, A.: Metamodeling mathematics: a precise and visual framework for describing semantics domains of UML models. In: Jézéquel, J.M., Hussmann, H., Cook, S. (eds.) UML 2002—The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany. Proceedings, Lecture Notes in Computer Science, vol. 2460, pp. 18–33. Springer, Heidelberg (2002)
48. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Reading (1998)
49. Ziemann, P.: An Integrated Operational Semantics for a UML Core Based on Graph Transformation. No. 14 in Monographs of the Bremen Institute of Safe Systems. Logos, Ph.D. Thesis (2006)
50. Ziemann, P., Hölscher, K., Gogolla, M.: On translating UML models into graph transformation systems. J. Vis. Lang. Comput. **17**(1), 78–105 (2006)

Author Biographies

Sabine Kuske is a lecturer at the Department of Computer Science at the University of Bremen in the north of Germany. She is a member of the Theoretical Computer Science Group headed by Hans-Jörg Kreowski. She received a Ph.D. in Computer Science in 2000 on a thesis entitled *Transformation units—a structuring principle for graph transformation systems*. Her research interests include all application areas of graph transformation, in particular, semantics and correctness aspects of autonomous systems, as well as visual modeling techniques such as UML.

Martin Gogolla is professor for Computer Science at University of Bremen, Germany and is the head of the Research Group Database Systems. His research interests include software development with object-oriented approaches, formal methods in system design, semantics of languages, and formal specification. Before joining University of Bremen he worked for the University of Dortmund and the Technical University of Braunschweig. His professional activities include: teaching computer science; publications in journals and conference proceedings; publication of two books; speaker to university and industrial colloquia; referee for journals and conferences; organizer of workshops and conferences (e.g., the UML conference); member in international and national program committees; contributor to international computer science standards (OCL 2.0 as part of UML 2.0).

Hans-Jörg Kreowski is professor for Theoretical Computer Science at the University of Bremen since 1982. He received his Ph.D. and his habilitation from the Technical University of Berlin where he was researcher and assistant professor from 1974 to 1982. He spent a sabbatical at IBM Research Center Yorktown Heights and was the first chair of the IFIP WG 1.3 (*Foundations of Systems Specification*). He is currently team leader in the Collaborative Research Centre 637 on *Autonomous Cooperating Logistic Processes*. His main research areas are algebraic specification, graph transformation, picture generation, theory of concurrency, and formal methods in software engineering and logistics.

Paul Ziemann is a former member of the Database Systems Group headed by Martin Gogolla at the University of Bremen. He received a PhD in Computer Science in 2006 on a thesis entitled *An Integrated Operational Semantics for a UML Core Based on Graph Transformation*. Since 2006, he has been developing object-oriented enterprise applications for a software company in Bremen.