Electronic Notes in Theoretical Computer Science 44 No. 4 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume44.html` 15 pages

GRACE as a unifying approach to graph-transformation-based specification [★]

Hans-Jörg Kreowski, Giorgio Busatto, Sabine Kuske

University of Bremen [1]

Abstract

In this paper, we sketch some basic ideas and features of the graph-transformation-based specification language GRACE. The aim of GRACE is to support the modeling of a wide spectrum of graph and graphical processes in a structured and uniform way including visualization and verification.

1 Introduction

Eventually, GRACE will be the acronym of a GRAph and rule CEntered specification language. Meanwhile, it is rather naming a creative process that is going on for some years and involves a varying number of researchers from Berlin, Bremen, Erlangen, München, Oldenburg, and Paderborn. At several working meetings, various aspects of graph transformation have been discussed like structuring, hierarchical graph models, object orientation, tool support, and graph exchange formats. The aims of these meetings have been to develop graph transformation as a methodology for the modeling and the specification of data-processing systems and to identify the necessary means to make graph transformation more visible in the applied areas of computer science. The outcome of the GRACE process consists so far of some papers which are more or less influenced by the discussed ideas and a part of which refers explicitly to GRACE (see, e.g., [10], [17], [11], [21], [16], [1], [8], [12], [13], [2], [4], [9], [14], [15], [18]).

In this paper, we introduce and survey GRACE as a unifying approach to graph-transformation-based specification with emphasis on uniform modeling of graphical processes. In particular, the following fundamentals are discussed:

- approach independence,

[★] This work was partially supported by the ESPRIT Working Group Applications of Graph Transformation (APPLIGRAPH) and the EC TMR Network GETGRATS (General Theory of Graph Transformation Systems).

[1] E-mail: {kreo, giorgio, kuske}@informatik.uni-bremen.de

- transformation units,
- modularization,
- visualization and animation,
- verification.

To show the usefulness of the GRACE concepts for handling graphical processes, we illustrate our considerations by Petri nets, finite state automata and several graph processing examples. In particular, we introduce new concepts for the transformation of graphs of different types into each other.

2 Approach Independence

In contrast to strings and trees, graphs are quite generic structures which are used in dozens of variants, types, and modes. Accordingly, one encounters quite a spectrum of competing graph transformation approaches in the literature (see, e.g., the three volumes of the Handbook on Graph Grammars and Computing by Graph Transformation [20], [5], [6]). While those familiar with graph transformation may appreciate the flexibility of graph notions and the wide range of choices to deal with them, others who want to use graph transformation for the first time may get confused easily. Approach independence is intended to avoid such a trouble by considering graph transformation as a uniform framework. This is achieved by the abstract and kind of axiomatic notion of a graph transformation approach that can be instantiated however it may be appropriate. A graph transformation approach provides a class of graphs, a class of rules, and a rule application operator specifying how a graph is directly derived from a graph by applying a rule. Since a rule defines a binary relation on graphs, a set of rules specifies a derivation relation as the reflexive and transitive closure of the union of the rule application relations. Moreover, a graph transformation approach provides means to restrict the nondeterminism of rule applications and their iteration. To illustrate the concepts, we consider Petri nets, finite state automata, hierarchical graphs and others.

2.1 Graph Transformation Approach

A graph transformation approach provides the basic ingredients of a GRACE program, namely graphs, rules, a rule application operator, graph class expressions, and control conditions. More formally, a *graph transformation approach* consists of a class $\mathcal{G}$ of *graphs*, a class $\mathcal{R}$ of *rules*, a *rule application operator* $\Rightarrow$ yielding a binary relation $\Rightarrow_r \subseteq \mathcal{G} \times \mathcal{G}$ for every $r \in \mathcal{R}$, a class $\mathcal{E}$ of *graph class expressions* such that every $e \in \mathcal{E}$ specifies a subclass $SEM(e) \subseteq \mathcal{G}$, and a class $\mathcal{C}$ of *control conditions* such that each $c \in \mathcal{C}$ specifies a binary relation $SEM(c) \subseteq \mathcal{G} \times \mathcal{G}$.

Examples for graph classes are labeled directed graphs, hypergraphs, trees, forests, finite automata, Petri nets, etc. Rule classes may vary from the more restrictive ones, like edge or node replacement to the more general ones as double pushout rules, single pushout rules, or PROGRES rules. In general, the rule application operator $\Rightarrow$ describes how the rules in $\mathcal{R}$ are applied to the graphs in $\mathcal{G}$.

The control conditions are used to regulate the derivation process. They determine, for example, the order in which rules may be applied. The aim of graph class expressions is to restrict the set of graphs to which certain rules may be applied or to filter out a subset of all the graphs that can be derived by a set of rules. Typically, a graph class expression may be some logic formula describing a graph property like connectivity, or acyclicity, or the occurrence or absence of certain node or edge labels.

Confer 2.4 to see how the components of a graph transformation approach are used.

2.2 Petri Nets

The usual firing of a transition in a place/transition net may be seen as a rule application. In this way, one gets a graph transformation approach $\mathcal{PT}$ that provides the basic elements of Petri nets. The class of graphs of $\mathcal{PT}$ consists of all *marked place/transition nets* $N = (S, T, F, m)$ where S is the set of *places*, T is the set of *transitions*, $F \subseteq S \times T \cup T \times S$ is the *flow relation* and $m : S \longrightarrow \mathbb{N}$ is a *marking.* The class of rules consists of all transitions. And such a rule t can be applied to a net $N = (S, T, F, m)$ if $t \in T$ and all input places of t carry tokens, i.e. $m(s) \geq 1$ for all $s \in {}^{\bullet}t$, or ${}^{\bullet}t \leq m$ for short. The resulting net differs from N only in the marking. The new marking is obtained by removing a token from each input place of t and adding a token to each output place, i.e. $m - {}^{\bullet}t + t^{\bullet}$ as usual for place/transition nets. Figure 1 shows the firing of a transition, where the rule application is denoted by $[\ >$ rather than $\Rightarrow$.

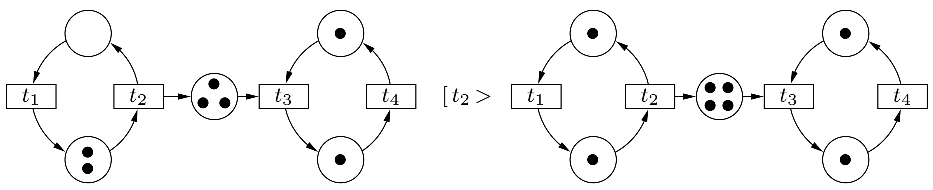

Fig. 1. Petri net: Firing t_2.

Similarly, one may choose multisets of transitions as rules yielding the graph transformation approach $\mathcal{PT}_+$. Again this is illustrated by an example (see Figure 2).

See 3.2 for the graph class and control conditions in the context of Petri nets.

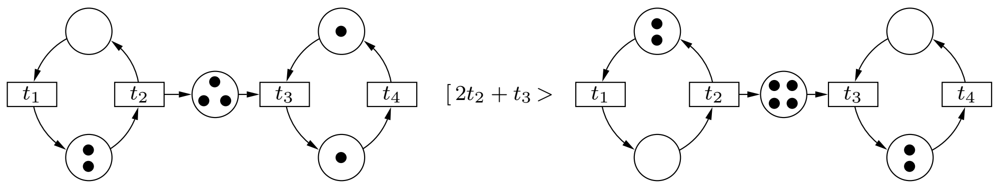

Fig. 2. Petri net: Firing $2t_2 + t_3$.

2.3 Finite State Automata

The recognition process of finite state automata is based on state transitions while reading input symbols. Such basic steps can be seen as rule applications of the graph transformation approach $\mathcal{FSA}$.

The class of graphs of $\mathcal{FSA}$ consists of pairs of input strings and state graphs. An input string may be represented by a string graph of the form depicted in Figure 3.

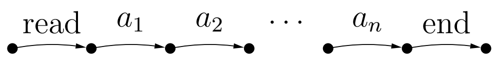

Fig. 3. A string graph.

where $a_1 \cdots a_n$ is the represented string and *read* and *end* are extra labels to mark the ends of a string. A state graph is an edge-labeled directed graph where each a-labeled edge represents a transition from the source node to the target node with input a. Moreover, there are an edge labeled with *init*, another labeled with *actual*, and some edges labeled with *fin* indicating the initial state, the actual state and the final states respectively. An $\mathcal{FSA}$ graph is depicted in Figure 4.

In Figure 5, an $\mathcal{FSA}$ rule is depicted. It can be applied to an $\mathcal{FSA}$ graph wherever the left-hand side occurs as subgraph. The application consumes the first input symbol and resets the actual state from the source to the target of the transition. The application of the rule with $x = c$ to the graph above is depicted in Figure 6.

See 3.3 for the graph class and control conditions used in the context of finite state automata.

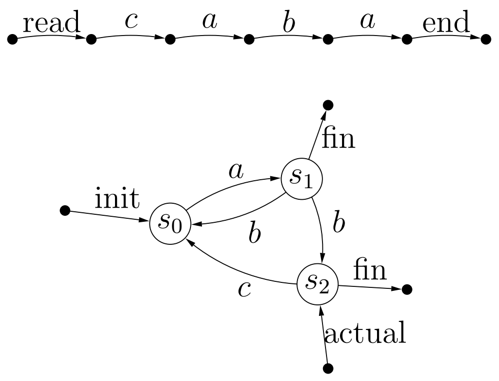

Fig. 4. An $\mathcal{FSA}$ graph.

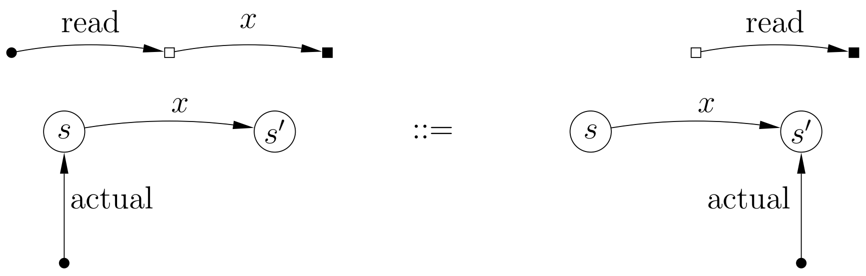

Fig. 5. An $\mathcal{FSA}$ rule.

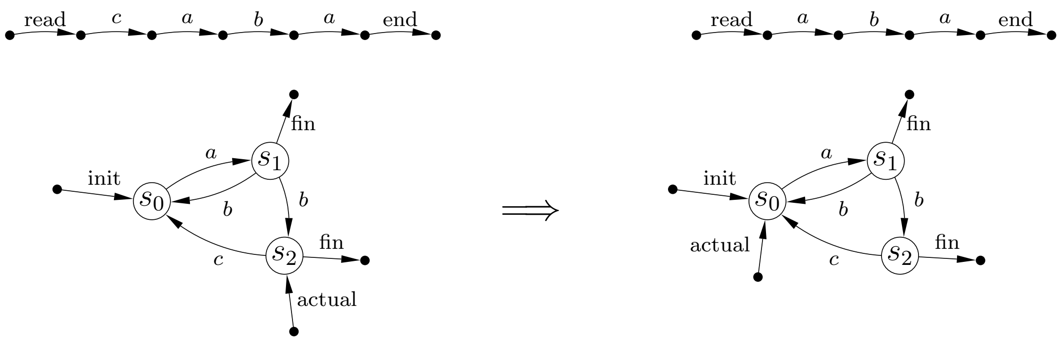

Fig. 6. Application of the rule with $x = c$.

2.4 Graph Processing

Given a graph transformation approach, one can derive graphs from graphs by the repeated application of rules where all processed graphs stem from the same class. But how can graph problems be handled which involve graphs of different types like the transformation of directed graphs into undirected ones or of hypergraphs into bipartite graphs? Graph transformations like these occur often in connection with the reduction of some graph problem into another graph problem like a connectivity test for directed graphs that can be reduced to a connectivity test for undirected graphs.

The problem of transforming graphs of different types into each other can be solved by combining two graph transformation approaches in such a way that the resulting approach covers the given approaches suitably. An explicit construction of this kind works as follows.

Let $\mathcal{A}_i = (\mathcal{G}_i, \mathcal{R}_i, \Longrightarrow, \mathcal{E}_i, \mathcal{C}_i)$ for $i = 1, 2$ be two graph transformation approaches such that any graph $G \in \mathcal{G}_i$ is provided with a set of distinguished items X_G, which may be the set of nodes, the set of edges, a particular subset of the set of nodes or edges or whatever. Analogously, we assume that each rule $r \in \mathcal{R}_i$ is also provided with such a set X_r and that each rule application $G \Rightarrow_r G'$ is associated with an occurrence map $f \colon X_r \to X_G$.

Then one can couple $\mathcal{A}_1$ an $\mathcal{A}_2$ into a new graph transformation approach $\mathcal{A}_1 \otimes \mathcal{A}_2$ which restricts the Cartesian product of $\mathcal{A}_1$ and $\mathcal{A}_2$ to those pairs of graphs and pairs of rules with equal distinguished items and to those rule applications that use the same occurrence map and keep the distinguished items invariant.

$$\mathcal{A}_1 \otimes \mathcal{A}_2 = (\mathcal{G}_1 \otimes \mathcal{G}_2, \mathcal{R}_1 \otimes \mathcal{R}_2, \Longrightarrow, \mathcal{E}_1 \times \mathcal{E}_2, \mathcal{C}_1 \times \mathcal{C}_2)$$

with

- $\mathcal{G}_1 \otimes \mathcal{G}_2 = \{(G_1, G_2) \in \mathcal{G}_1 \times \mathcal{G}_2 \mid X_{G_1} = X_{G_2}\}$,
- $\mathcal{R}_1 \otimes \mathcal{R}_2 = \{(r_1, r_2) \in \mathcal{R}_1 \times \mathcal{R}_2 \mid X_{r_1} = X_{r_2}\}$,
- $(G_1, G_2) \underset{(r_1,r_2)}{\Longrightarrow} (G_1', G_2')$ for $(G_1, G_2), (G_1', G_2') \in \mathcal{G}_1 \otimes \mathcal{G}_2$, $(r_1, r_2) \in \mathcal{R}_1 \otimes \mathcal{R}_2$ provided that $G_i \underset{r_1}{\Longrightarrow} G_i'$, $X_{G_i} = X_{G_i'}$ for $i = 1, 2$, and $f_1 = f_2$ for the corresponding occurrence maps.
- $\mathit{SEM}(e_1, e_2) = \mathit{SEM}(e_1) \times \mathit{SEM}(e_2) \cap \mathcal{G}_1 \otimes \mathcal{G}_2$ for $(e_1, e_2) \in \mathcal{E}_1 \times \mathcal{E}_2$,
- $((G_1, G_2), (G_1', G_2')) \in \mathit{SEM}(c_1, c_2)$ iff $(G_1, G_2), (G_1', G_2') \in \mathcal{G}_1 \otimes \mathcal{G}_2$ and $(G_i, G_i') \in \mathit{SEM}(c_i)$ for $c_i \in \mathcal{C}_i$ and $i = 1, 2$.

If each set X of distinguished items is associated with two particular graphs $\mathit{graph}_i(X) \in \mathcal{G}_i$ for $i = 1, 2$ with $X_{\mathit{graph}_i(X)} = X$, then one can identify every graph $G_1 \in \mathcal{G}_1$ with $(G_1, \mathit{graph}_2(X_{G_1}))$ and $G_2 \in \mathcal{G}_2$ with $(\mathit{graph}_1(X_{G_2}), G_2)$. In this sense, we have $\mathcal{G}_1 \cup \mathcal{G}_2 \subseteq \mathcal{G}_1 \otimes \mathcal{G}_2$. Moreover, we may use $\mathcal{G}_1$ and $\mathcal{G}_2$ as extra graph class expressions with $\mathit{SEM}(\mathcal{G}_i) = \mathcal{G}_i$ for $i = 1, 2$ (cf. 3.4), i.e. that we use the names of the graph classes to refer to them in the coupled approach where they are subclasses of the class of coupled graphs. To avoid confusion between the classes of graphs and their names, one may use other suitable identifiers that $\mathcal{G}_1$ and $\mathcal{G}_2$.

A typical example of such a coupling is given by approaches $\mathcal{A}_{\mathit{dir}}$ and $\mathcal{A}_{\mathit{undir}}$ with classes of directed graphs, $\mathcal{G}_{\mathit{dir}}$, and of undirected graphs, $\mathcal{G}_{\mathit{undir}}$, respectively, where the sets of nodes are considered as distinguished, i.e. $X_G = V_G$ for $G \in \mathcal{G}_{\mathit{dir}} \cup \mathcal{G}_{\mathit{undir}}$. To simplify the technicalities, we restrict our consideration to simple graphs meaning that the set of edges E_G of $G \in \mathcal{G}_{dir}$ is a subset of $V_G \times V_G$ and the set of edges E_G of $G \in \mathcal{G}_{undir}$ is a subset of the 1-element subsets and the 2-elements subsets of V_G. Moreover, we consider only rules of the form $r = (L ::= R)$ where L and R are graphs with $X_r = V_L = V_R$. Given an occurrence map $f\colon X_r \to X_G$, the application of r to G yields the graph G' with $V_{G'} = V_G$ and $E_{G'} = (E_G - f(E_L)) \cup f(E_R)$. Here the image of a set of edges is defined by mapping the nodes of each edge. This definition applies to directed and undirected graphs as well. Accordingly, $(G_1, G_2) \in \mathcal{G}_{dir} \otimes \mathcal{G}_{undir}$ is a pair of a directed and an undirected graph with the same set of nodes. As the rule applications are only allowed if the distinguished items are kept invariant, only the sets of edges are rewritten. Moreover, if a graph – directed or undirected – is paired with its set of nodes considered as discrete graph with empty set of edges, the class of directed graphs as well as the class of undirected graphs can be considered as subclasses of the class of coupled graphs. Accordingly, we will use the terms *directed graphs* and *undirected graphs* as graph class expressions. The first expression specifies all pairs $(G, G') \in \mathcal{G}_{dir} \otimes \mathcal{G}_{undir}$ such that G' is a discrete graph whereas $\mathit{SEM}(\mathit{undirectedgraphs})$ contains all pairs $(G, G') \in \mathcal{G}_{dir} \otimes \mathcal{G}_{undir}$ such that G is a discrete graph. Additionally, we will use one particular control condition, which will be explained in Section 4. The discussion of this approach is continued in 3.4.

2.5 Hierarchical Graphs

Using a similar coupling mechanism, we have introduced a framework for hierarchical graphs in [3]. Graphs with distinguished sets of nodes from an arbitrary graph are combined with acyclic graphs and bipartite connection graphs. A connection graph relates the distinguished set of nodes with the nodes of the acyclic graph and imposes a hierarchy on the graph in this way.

3 Transformation Units

Transformation units are the main syntactic entities of GRACE that allow to specify binary relations on graphs. They consist of rules, input, output and control conditions as well as of import components. Semantically, such a transformation unit imports binary relations on graphs which are interleaved with the rule applications such that input, output and control conditions are obeyed. In this way, a transformation unit encapsulates a computational process. And since it specifies a binary relation on graphs, it can be imported by other transformation units such that the concept supports structuring in a simple, but effective way. Based on the graph transformation approach $\mathcal{PT}$ of 2.2, each place/transition net with an initial marking defines a transformation unit the semantic relation of which comprises all reachable markings. Given the graph transformation approach $\mathcal{FSA}$ of 2.3, each finite state automaton defines a transformation unit the semantic relation of which describes the recognized language.

Moreover, we illustrate the use of the coupling mechanism as introduced in 2.4.

3.1 Syntax and Semantics of Transformation Units

Let $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Longrightarrow, \mathcal{E}, \mathcal{C})$ be a graph transformation approach.

A *transformation unit* (over $\mathcal{A}$) is a system $tu = (I, U, P, C, T)$ where I and T are graph class expressions, U is a set of identifiers, P is a set of rules, and C is a control conditions.

The elements of $SEM(I)$ are called *initial graphs*, and the elements of $SEM(T)$ *terminal graphs* respectively. U is the *import component* of tu which is also called *use component*.

Let $SEM(t) \subseteq \mathcal{G} \times \mathcal{G}$ be some binary relation on graphs for each $t \in U$ and $SEM(U)$ the union of these relations. Then the *interleaving semantics* $SEM(tu)$ of tu is defined by

$$(\underset{P}{\Longrightarrow} \cup SEM(U))^* \cap SEM(I) \times SEM(T) \cap SEM(C)$$

i.e. initial and terminal graphs are related by interleaving rule applications and calls of imported relations while the control condition is obeyed.

3.2 Petri Nets as Transformation Units

Consider the graph transformation approach $\mathcal{PT}$ of place/transition nets, and let $N = (S, T, F, m)$ be a marked net. Then N induces a transformation unit $tu(N) = (N, \emptyset, T, \mathit{true}, \mathit{all})$ where N as a graph class expression specifies itself, i.e. $SEM(N) = \{N\}$, and the transitions of N are available as rules. Moreover, nothing is imported, there is no extra control, and all marked nets are considered as terminal. Therefore, the semantics is given by all firing sequences which start in the initial marking such that the semantic relation of $tu(N)$ corresponds to the set of reachable markings.

3.3 Finite State Automata as Transformation Units

Consider the graph transformation approach $\mathcal{FSA}$, and let A be a finite state automaton with input alphabet I. Then A induces a transformation unit $tu(A) = (I_A, \emptyset, P_A, \mathit{true}, T_A)$ where I_A specifies all $\mathcal{FSA}$-graphs that consist of input strings and the state graph of A with the initial state as actual state, P_A consists of all rules corresponding to state transitions of A, and T_A specifies the $\mathcal{FSA}$-graphs that consist of the empty input string and the state graph of A with one of the final states as actual state. Therefore, an initial and a terminal graph are semantically related if and only if the initial input string is accepted by A. In this way, the recognition process of a finite state automaton can be seen as a special case of the computational processes provided by transformation units.

3.4 Transforming directed graphs into undirected graphs

Based on the graph transformation approach $A_{dir} \otimes A_{undir}$ in 2.4, the following transformation unit specifies the transformation of directed graphs into undirected graphs. Starting from a directed graph, each of its directed edges is replaced by an undirected edge between the same nodes as long as one ends up with an undirected graph.

dir-undir

initial: *directed graphs*

rules: $\bullet\!\!\longrightarrow\!\!\blacksquare \;::=\; \bullet\!\!-\!\!\!-\!\!\!-\!\!\blacksquare$

terminal: *undirected graphs*

The given rule represents the combination of the rule $\bullet\!\!\longrightarrow\!\!\blacksquare \;::=\; \bullet \quad \blacksquare \;\in \mathcal{R}_{dir}$, which removes a directed edge, with $\bullet \quad \blacksquare \;::=\; \bullet\!\!-\!\!\!-\!\!\!-\!\!\blacksquare \;\in \mathcal{R}_{undir}$, which adds an undirected edge between two nodes. The two nodes are shared by both rules such that the removed and added edge are incident to the same nodes. If one starts the application from a directed graph, all its edges must be replaced by undirected ones before the transformation results in the corresponding undirected graph.

In a similar way, one can specify a transformation unit that transforms a hypergraph into a bipartite graph where each hyperedge is represented by a new node and many other transformations between graphs of different types.

4 Modularization

In this section, we summarize the module concept of GRACE briefly. More details can be found in [12], [4], [9], [14], [15].

A transformation unit specifies a binary relation on graphs depending on the choice of the used binary relations in the import part. But how can the used relations be chosen? One possibility is that relations are stored in some relation library and may be called by their names. Another possibility is to use the relations specified by transformation units. Both together leads to the notion of a transformation module as a network of transformation units which use each other and where the edges reflect the use structure. Moreover, each module has an import interface, which plays the same role as the import component of units, and an export interface containing the units that are provided by the module to the environment and may be imported by other units and modules later on. More formally, a *transformation module* is a system $MOD = (IMPORT, BODY, EXPORT)$ where $BODY$ is a set of transformation units, $IMPORT$ is a set of identifiers, and $EXPORT \subseteq BODY \cup IMPORT$. Moreover, every body unit may use only units of $BODY$ and $IMPORT$, i.e. for every $tu = (I, U, P, C, T) \in BODY$, we require that $U \subseteq BODY \cup IMPORT$. (Note that we do not explicitly distinguish transformation units and their names.)

Semantically, a transformation module specifies a set of binary relations on graphs, one relation for each export unit, depending on the choice of the imported relations. If the network underlying a module is acyclic (and finite), the semantics is constructed level by level by means of the interleaving semantics. One starts with the transformation units that use relations from the import interface only, and proceeds step by step with those units whose use component has got a semantics already. In the case, of cyclic use structures, the interleaving of rule applications and used relations is repeated ad infinitum starting from the imported relations and the empty relations for the body units.

To illustrate the module concept, we present a transformation module which provides connectivity tests for directed and undirected graphs. It imports a connectivity test for undirected graphs and the unit *dir-undir* in 3.4.

connectivity-tests

import:	*connectivity-test-undir, dir-undir*
body:	*connectivity-test-dir*
export:	*connectivity-test-dir, connectivity-test-undir*

with

connectivity-test-dir

uses:	*connectivity-test-undir, dir-undir*
conds:	*dir-undir; connectivity-test-undir.*

This transformation unit specifies the connectivity test for directed graphs as the sequential composition of the transformation from directed to undirected graphs followed by the corresponding test for undirected graphs. The latter test may be available in some library in which form ever, or it may also be specified as a transformation unit like the following.

connectivity-test-undir

initial:	*undirected graphs*
rules:	●——■ ::= ● ■ | ●
terminal:	●

This unit has got two rules with the same left-hand side and two alternative right-hand sides. The first rule removes an edge if applied in the way defined in 2.4. The second rule is an extra rule not yet introduced. It can be applied to a node with degree one such that this node and its only incident edge is removed.

5 Visualization and Animation

Graphical structures like diagrams, maps, nets, and graphs are very popular because they allow one to represent complex structures in an intuitive way GRACE is intended to support visual modeling by means of a graphical interface. This interface must be extremely flexible because of the approach independence. There is no fixed notion of graphs, but the graphs of a graph transformation approach can be chosen according to one's tastes or needs. And the graphical interface should respect and reflect this generality by offering a wide spectrum of possibilities. At least, it should be feasible to deal with directed graphs as well as with undirected ones, with hypergraphs as well as with simple graphs, with labeled graphs as well as with unlabeled ones etc. The most important requirement is that the visualization of graphs can be done in such a way that the graphs look like the structures they represent. For example, a Petri net should look like a Petri net, and a UML class diagram should look like a UML class diagram (see e.g. Figure 7). One may even imagine that in modeling the production processes of a factory by the use of graph transformation the representing graphs look like a machine hall. For purposes like this, the graphical interface of GRACE may offer colored 3-D representations of graphs (as the experimental implementation of GRACEland by Martin Faust [7]).

If the graphs can be properly visualized in the GRACE context, then one has also a visualization of a rule application, which is given by two graphs. Therefore, animation is obtained by the iteration of rule applications such that the dynamic behaviour of transformation units can be looked at. This applies even to transformation units that import other transformation units. If the semantic process of the imported units can be animated, one gets the animation of the importing units by interleaving the visualized rule applications with the animation sequences of the imported units.

6 Verification

The GRACE concepts support verification in two ways. On one hand, the structuring by units and modules gives rise to structured proofs. On the other hand, the interleaving semantics provides an induction proof schema.

To illustrate structured proofs in a very simple case, consider the unit *connectivity-test-dir* in Section 4. By definition, the relation it specifies is the sequential composition of the relations specified by *dir-undir* in 3.4 and *connectivity-test-undir*. If the latter is correct meaning that it tests the connectivity of undirected graphs, and if the former transforms a directed into an undirected graph by forgetting the direction of edges, but keeping the incidences, then the sequential composition is a correct realization of a connectivity test for directed graphs as it is often defined in graph theory.

To show the correctness of *connectivity-test-undir* and *dir-undir*, one can

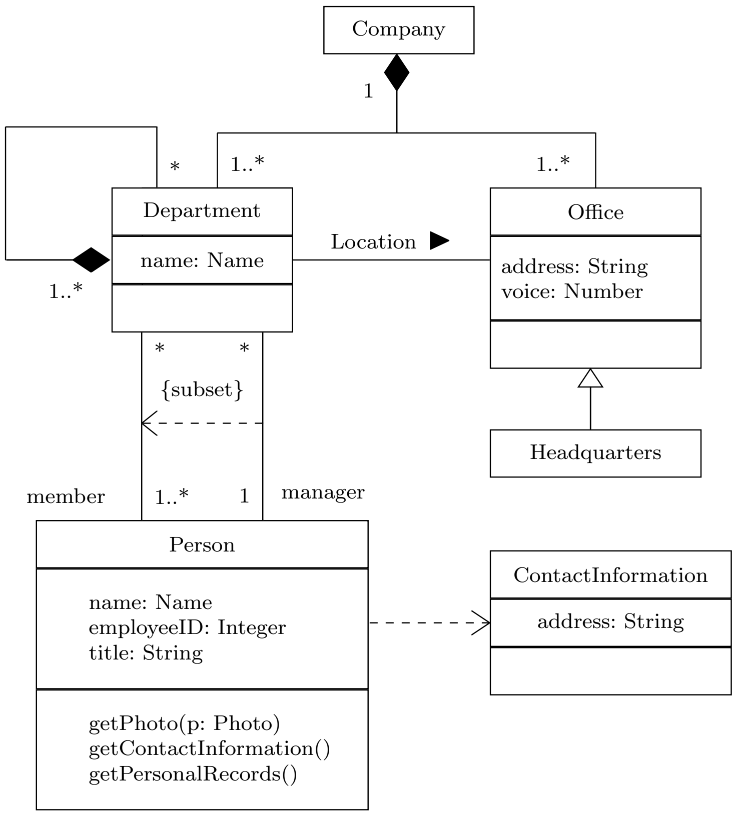

Fig. 7. An UML class diagram.

run induction proofs on the length of interleaving sequences, which are derivation sequences in the two cases because both units do not use other units.

In the first case, it is easy to prove the following: If $G \Rightarrow^* G'$ and G' is connected, then G is connected. Hence, only connected graphs can be derived into a single node. Moreover, one can prove by induction on the number of edges, that any connected graph can be derived into a single node. Both together means for the semantic relation of *connectivity-test-undir* that a graph G is related to the single-node graph if and only if G is connected. In this sense, we have a correct connectivity test for undirected graphs.

Again by induction on the lengths of derivation sequences, one can prove that the incidences are kept invariant by derivations in *dir-undir*. Moreover, the rule is applicable as long as there are directed edges. This proves the transformation of directed into undirected graphs as desired.

Altogether, the module *connectivity-tests* in Section 4 provides correct connectivity tests for directed and undirected graphs. Further and more complicated examples of correctness proofs for GRACE units can be found in

[12,13,19].

7 Conclusion

In this paper, we have sketched some basic ideas and features as well as some potentials of the graph-transformation-based language GRACE. It is approach-independent meaning that the type of graphs, rules, rule applications, etc. is not fixed from the very beginning, but can be chosen according to the intended application and one's taste. In particular, the structuring concepts of transformation units and transformation modules work quite unrestricted while special constructions like the coupling of approaches require special assumptions.

In contrast to syntax and semantics of the structuring concepts, which are already discussed and worked out in some detail (cf. [16], [12], [13],[4], [9]), the ideas how to deal with graphs of different types and how to visualize, animate, and verify graph transformation in GRACE are in a very premature state and need further investigation.

References

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.

[2] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. UML 2000 - The Unified Modeling Language. Advancing the Standard*, volume 1939 of *Lecture Notes in Computer Science*, pages 294–308, 2000.

[3] Giorgio Busatto, Hans-Jörg Kreowski, and Sabine Kuske. An abstract hierarchical graph data model. In preparation, 2001.

[4] Frank Drewes, Peter Knirsch, Hans-Jörg Kreowski, and Sabine Kuske. Graph transformation modules and their composition. In Manfred Nagl, Andreas Schürr, and Manfred Münch, editors, *Proc. Applications of Graph Transformations With Industrial Relevance*, volume 1779 of *Lecture Notes in Computer Science*, pages 15–30, 2000.

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

[6] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph*

Transformation, Vol. 3: Concurrency, Parallelism, and Distribution. World Scientific, Singapore, 1999.

[7] Martin Faust. The GRACEland web page. URL: `http://www.informatik.uni-bremen.de/theorie/GRACEland`, 1998.

[8] Reiko Heckel, Gregor Engels, Hartmut Ehrig, and Gabriele Taentzer. Classification and comparison of module concepts for graph transformation systems. In Ehrig et al. [5], pages 639–689.

[9] Reiko Heckel, Berthold Hoffmann, Peter Knirsch, and Sabine Kuske. Simple modules for GRACE. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 383–395, 2000.

[10] Hans-Jörg Kreowski. Graph grammars for software specification and programming: An eulogy in praise of GRACE. In Francesc Rosselló Llompart and Gabriel Valiente Feruglio, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science*, Technical Report, Palma de Mallorca, pages 55–61, 1995.

[11] Hans-Jörg Kreowski and Sabine Kuske. On the interleaving semantics of transformation units — a step into GRACE. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 89–108, 1996.

[12] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. In Ehrig et al. [5], pages 607–638.

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

[14] Hans-Jörg Kreowski and Sabine Kuske. Note on approach-independent structuring concepts for rule-based systems. In Hartmut Ehrig and Gabi Taentzer, editors, *Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, Technical Report Nr. 2000-2, Technische Universität Berlin, pages 41–49, 2000.

[15] Hans-Jörg Kreowski and Sabine Kuske. Suggestions on the modularization of rule-based systems. In Martin Wirsing, Martin Gogolla, Hans-Jörg Kreowski, Tobias Nipkow, and Wolfgang Reif, editors, *Proc. Rigorose Entwicklung software-intensiver Systeme*, Technical Report 0005, University of Munich, pages 73–82, 2000.

[16] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering*, 7(4):479–502, 1997.

[17] Sabine Kuske. Semantic aspects of the graph and rule centered language GRACE. In Francesc Rosselló Llompart and Gabriel Valiente Feruglio, editors,

Proc. Colloquium on Graph Transformation and its Application in Computer Science, Technical Report, Palma de Mallorca, pages 63–70, 1995.

[18] Sabine Kuske. More about control conditions for transformation units. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 323–337, 2000.

[19] Sabine Kuske. *Transformation Units – A Structuring Concept for Graph Transformation Systems.* PhD thesis, Universität Bremen, 2000.

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

[21] Andy Schürr. Programmed graph transformations and graph transformation units in GRACE. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 122–136, 1996.