Simple Modules for GRACE*

Reiko Heckel\(^1\), Berthold Hoffmann\(^2\), Peter Knirsch\(^2\), and Sabine Kuske\(^2\)

\(^1\) Fachbereich 17 Informatik
Universität-GH Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
reiko@uni-paderborn.de

\(^2\) Fachbereich 3 Mathematik - Informatik
Universität Bremen
Postfach 33 04 40
D-28334 Bremen, Germany
{hof,knirsch,kuske}@informatik.uni-bremen.de

Abstract. The language GRACE is being proposed for specifying and programming in a graph-centered environment based on graph transformation. Emphasis in the design of GRACE is laid on modular structuring of programs. In this paper we present a simple kind of modules for the language which encapsulate rules and graph transformation units, and export some of them. The local transformation units define compound graph transformations that call rules and other transformation units which are either defined by the surrounding module, or imported from other modules. The interleaving semantics of modules specifies a binary relation on graphs for each exported item.

1 Introduction

GRACE is a graph transformation language for specifying and programming in a graph-centered environment that is being developed by researchers at TU Berlin, Universität Bremen, Universität Erlangen, Universität der Bundeswehr München, Universität Oldenburg, and Universität Paderborn. In contrast to other graph transformation languages like PROGRES [Sch94] and AGG [TB94], GRACE is approach-independent in that it may host different notions of graph transformation—in a single program eventually.

GRACE shall extend the abstract notion of graph transformation by concepts for programming and specification that support the development of large-scale systems. As a first step, transformation units were introduced in [KK96] to specify control, functional composition, and encapsulation (see also [KKS98, AEH\(^+\)99]). In this paper, we relieve transformation units from their interfaces and local rules, and introduce modules with export and import interfaces that encapsulate transformation units and rules. The resulting structure is

\* 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).
similar to that of procedures and packages in Ada [Ada95], or of functions and modules in Haskell [PHA+97].

This kind of modules will not suffice alone: we also need a concept similar to classes in object-oriented languages, for data-driven encapsulation. However, a simple module concept will also be useful on top of such a rather “fine-grained” class concept, see Java [AG95].

The paper is organized as follows: the approach-independence of GRACE is explained in the following section. Afterwards, we introduce graph transformation units which, together with graph transformation rules, constitute the basic components of modules. In section 4 graph transformation modules are presented together with their operational semantics. Section 5 shows how structured graph transformation systems can be built up from a set of modules resulting in a module system. The usefulness of this concept is illustrated with a small example from the area of graph algorithms. We conclude by presenting related work and outlining how graph transformation modules can be extended with respect to various aspects.

Throughout this paper, ID shall denote a vocabulary of identifiers that is used to name entities like rules, transformations, and modules. If $N \subseteq ID$ is a subset of names, a named set (over $N$) is a mapping $NS : N \rightarrow S$ into some given set $S$. We write $n : s \in NS$ to refer to some $n \in N$ with $NS(n) = s$.

2 Approach-Independence

There are many ways to define graph transformation (see [Roz97] for a survey). All of them allow some class of graphs to be transformed by applying some kind of rules. In most cases, these rules can be applied in arbitrary order and frequency; some notions of programmed graph transformation also allow to specify conditions on graphs, and to control the application of rules.

Since GRACE is approach-independent, we just specify the requirements for graph transformation approaches and make the language generic with respect to this specification.

Definition 1. ([KK96]) A graph transformation approach is a system

$$A = \langle \mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C} \rangle$$

with a class $\mathcal{G}$ of graphs, a class $\mathcal{R}$ of graph transformation rules, a rule application operator $\Rightarrow$, a class $\mathcal{E}$ of graph class expressions, and a class $\mathcal{C}$ of control conditions, where

- $\Rightarrow$ defines a graph transformation relation $\Rightarrow, \subseteq \mathcal{G} \times \mathcal{G}$ for every rule $r \in \mathcal{R}$,
- each $X \in \mathcal{E}$ specifies a graph language $\mathcal{L}(X) \subseteq \mathcal{G}$, and
- each $C \in \mathcal{C}$ specifies a graph transformation relation $\text{SEM}_E(C) \subseteq \mathcal{G} \times \mathcal{G}$, for every graph transformation environment $E : \text{ID} \rightarrow 2^{\mathcal{G}}$, a mapping that associates transformation relations with names which may occur in $C$. 

\footnote{For a set $A$, $2^A$ denotes its power set.}
For each control condition $C \in C$, $\text{names}(C) \subseteq ID$ denotes the set of names occurring in $C$.

**Example 1.** As an example we define a simple graph transformation approach which will be used throughout this paper.

- The class of graphs consists of all undirected graphs without multiple edges labeled over an alphabet $\Sigma$ with $\ast \in \Sigma$. The symbol $\ast$ is a special one standing for **unlabeled**.

  Formally, each graph $G \in \mathcal{G}$ is a system $(V, E, l, m)$ where $V$ is a finite set of **nodes**, $E \subseteq V \times V$ is a finite set of **edges**, $l : V \rightarrow \Sigma$ and $m : E \rightarrow \Sigma$ are mappings associating a **label** to each node and each edge in $G$.

  A graph $G' = (V', E', l', m') \in \mathcal{G}$ is a **subgraph** of $G$ if $V' \subseteq V$, $E' \subseteq E$, $l' = l|V'$, and $m' = m|E'$.\(^2\)

  An occurrence $g$ of $G'$ in $G$ consists of two mappings $g_V : V' \rightarrow V$ and $g_E : E' \rightarrow E$ such that the label of each node and edge in $G'$ is preserved, i.e., $g_V \circ l = l'$ and $g_E \circ m = m'$.

- Each rule is of the form $r = (\mathcal{N}, L \rightarrow R)$ where $L$ and $R$ are graphs with equal sets of nodes and $\mathcal{N}$ is a set of graphs such that for each $M \in \mathcal{N}$, $L$ is a proper subgraph of $M$. It is known as rule with negative context condition as described in [HHT96]. An example of such a rule is depicted below in pseudo code notation.

```
mark:       •.\ast./\m

\rightarrow \m
```

The rule *mark* consists of a left- and a right-hand side separated by an arrow. The left-hand side shows an abbreviated notation of a negative context condition meaning that the rule *mark* can be applied if there is neither an unlabeled nor an $m$-labeled edge between the nodes. This directly corresponds to the above introduced set $\mathcal{N} = \{ \text{--\m-\m-} \}$. If the negative context condition is satisfied an $m$-labeled edge is inserted.

- A rule $r = (\mathcal{N}, L \rightarrow R)$ is applied to a graph $G$ resulting in graph $G'$ according to the following steps: (1) **Choose** an occurrence $g$ of $L$ in $G$. (2) **Check** the negative context condition: for each $M \in \mathcal{N}$ there is no occurrence $h$ of $M$ in $G$ such that $h$ restricted to $L$ is equal to $g$. (3) **Remove** the image of each edge in $L$, i.e., for each $e \in E_L$ remove $g_E(e)$ (4) **Add** $R$ to the resulting graph by gluing each node $v$ in $R$ with its image $g_{V_L}(v)$ in $G$. Hence, the rule *mark* inserts an $m$-labeled edge between two existing nodes if they are neither connected by an unlabeled nor by an $m$-labeled edge.

* As graph class expressions we use the default expression *all* specifying the set of all graphs, as well as $F$ and $\neg F$ for all $F \subseteq \Sigma$. The expression $F$ specifies all graphs labeled over $F$ whereas $\neg F$ admits only graphs without labels from $F$.

\(^2\) For a function $f : A \rightarrow B$ and a set $C \subseteq A$ $f|C$ denotes the restriction of $f$ to $C$. 
Fig. 1. An application of the rule mark

- Typical examples of control conditions are regular expressions over ID and as-long-as-possible [Kus98]. Here, we combine both in the following way: for each $n \in ID$, $n!$ is a control condition which applies the rule or transformation unit named $n$ as long as possible, i.e. for each environment $E$, $\text{SEM}_E(n!) = \{(G, G') \in E(n)^* | \exists G'' \in G : (G', G'') \in E(n)\}$. Let $C' = \{n! | n \in ID\}$. Then the class of control conditions of our example approach consists of all regular expressions over $C' \cup ID$, which are recursively defined as follows: $C' \cup ID \in \text{REG}$, and $(C_1; C_2) \in \text{REG}$ if $C_1, C_2 \in \text{REG}$. Semantically, $n \in ID$ applies the rule or transformation unit $n$ exactly once, i.e. $\text{SEM}_E(n) = E(n)$ for each environment $E$; $(C_1; C_2)$ applies first $C_1$ and then $C_2$, i.e. $\text{SEM}_E((C_1; C_2)) = \text{SEM}_E(C_1) \circ \text{SEM}_E(C_2)$. For example, the condition (mark !; erase) where erase is a rule applies first mark as long as possible and then erase once.

Eventually, it shall be possible to define heterogeneous graph transformation systems, composed from modules defined in different approaches. For now, we stick to homogeneous systems specified in a single approach.

3 Graph Transformation Units

In a simple graph transformation system, rules define basic graph transformations that may be applied in arbitrary order and frequency. We introduce graph transformation units for defining compound graph transformations that call rules and other transformation units, by control conditions of the underlying approach. This provides for functional composition. Also, they allow pre- and post-conditions to be specified, by graph class expressions of the underlying approach. This allows to type units with respect to the graphs they transform. The rules and transformation units used in a graph transformation unit are defined
or imported by the surrounding module, and are used via the transformation environment in which the unit is interpreted.

**Definition 2.** A graph transformation unit over $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$ is a tuple $\text{trafo} = (I, C, T)$ where

- $I \in \mathcal{E}$ is an initial graph class expression,
- $C \in \mathcal{C}$ is a control condition, and
- $T \in \mathcal{E}$ is a terminal graph class expression.

Let $\mathcal{T}_\mathcal{A}$ denote the set of graph transformation units over $\mathcal{A}$.

The semantics of each transformation unit is a binary relation on graphs. It consists of each pair $(G, G')$ of graphs where $G$ is initial, $G'$ is terminal, $(G, G')$ is specified by the control condition, and $G'$ can be obtained from transforming $G$ with the rules and transformation units to which the names in the control condition refer. Hence, the semantics of a transformation unit is computed with respect to an environment which associates a semantics to each name occurring in the control condition.

**Definition 3 (semantics of graph transformation units).** Consider a graph transformation unit $\text{trafo} = (I, C, T)$ over $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$ and a graph transformation environment $E : \text{ID} \to 2^{\mathcal{G} \times \mathcal{G}}$. Then the semantics of $\text{trafo}$ in $E$, denoted by $\text{SEM}_E(\text{trafo})$, consists of all pairs $(G, G')$ of graphs such that

- $(G, G') \in \mathcal{L}(I) \times \mathcal{L}(T),$
- $(G, G') \in \text{SEM}_E(C),$ and
- $(G, G') \in \left( \bigcup_{n \in \text{names}(C)} E(n) \right)^*.$

**Remarks.**

1. The last point of the preceding definition means that for each $(G, G') \in \text{SEM}_E(\text{trafo})$ there is a sequence $G_0, \ldots, G_j$ of graphs such that (1) $G_0 = G$ and $G_j = G'$, and (2) for $i = 1, \ldots, j$ $(G_{i-1}, G_{i}) \in E(n)$ for some $n \in \text{names}(C)$. Hence, if $E$ associates to each name in $C$ the semantics of the rule or transformation unit it refers to, $G'$ is obtained from $G$ by interleaving rule applications with applications of transformation units.
2. Note that all control conditions of the class $\mathcal{C}$ in our example approach fulfill the property of the last item in the definition. But in general one may have other types of control conditions. A simple example is a pair of graphs without names, specifying itself.
3. The graph transformation units of [KK96] contain also a set $U$ of used (or imported) graph transformation units, and a set $R$ of local graph transformation rules. In our definition of graph transformation units, imports and rules belong to the surrounding graph transformation module, and are used via the environment.

**Example 2.** An example of a transformation unit is the `invert` where `unmark` and `erase` are names of rules.
\begin{center}
\begin{tabular}{|c|c|}
\hline
invert & \{+\} \\
initial & \{+\} \\
cond & mark! ; erase! ; unmark! \\
terminal & all \\
\hline
\end{tabular}
\end{center}

The unit \textit{invert} takes an unlabeled graph as its input, applies first the rule \textit{mark} as long as possible, then the rule \textit{erase} as long as possible, and finally the rule \textit{unmark} as long as possible. As terminal graphs all graphs are allowed. This transformation unit adds \textit{m}-labeled edges until the graph is complete. After that it deletes previously existing unlabeled edge and then finally unmarks the \textit{m}-labeled edges. Therefore \textit{invert} returns the complement of its input graph.

\section{Graph Transformation Modules}

A graph transformation module encapsulates a named set of graph transformation rules and graph transformation units, and exports some of them. It imports transformation units and rules from other modules. The semantics of a module is a set of graph transformations, i.e. a set of binary relations on graphs.

More precisely, a graph transformation module consists of an import part, a body, and an export part. The body defines a set of named local rules and transformation units. The export part consists of a subset of the names of local rules and transformation units. It determines which local units and rules can be imported by other modules. Hence, a local unit or rule cannot be used from the outside if its name is not contained in the export. The import part consists of a set of names (referring to rules and transformation units of other modules) which can be used in local transformation units. This means that each local unit may contain names from the import part in its control condition. All other names of the control condition must refer to other local rules or transformation units.

\textbf{Definition 4.} Let $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{E}, \mathcal{C})$. Then a triple $\text{mod} = \langle IM, DEF, EX \rangle$ where

- $IM \subseteq ID$ is a set of \textit{imported names},
- $DEF : N \rightarrow \mathcal{R} \cup \mathcal{T}_A$ is a finite named set of \textit{rules and transformation units} over some set $N \subseteq ID$ of names,
- $EX \subseteq N$ is a set of \textit{exported names},

is called a \textit{graph transformation module over} $\mathcal{A}$ if the following conditions are satisfied:

- No imported name is re-introduced for a local rule or transformation unit, i.e. $N \cap IM = \emptyset$.
- Transformation units use only names that are imported, or locally defined. More precisely, for all $\langle I, C, T \rangle \in DEF$, $\text{names}(C) \subseteq N \cup IM$.

Let $\mathcal{M}_\mathcal{A}$ denote the set of graph transformation modules over $\mathcal{A}$. 
For technical simplicity, we define the semantics for non-recursive modules in the following, wherein graph transformation units do not call themselves recursively. Formally, a module $M$ is non-recursive, if the relation $\sim^M$ on the set of transformation units in $M$ is acyclic, where $t : \langle I, C, T \rangle \sim^M t' : \langle I', C', T' \rangle$ if $t' \in \text{names}(C)$. Note, however, that the definition of the semantics below can be generalized to the case of recursive modules analogously to the generalization of the interleaving semantics of transformation units to the nested case (see [KKS98]).

The semantics of a module $mod$ associates a binary relation on graphs to each exported name. It is defined as the restriction of the so-called environment of $mod$ to its export part. This module environment maps a binary relation on graphs to each locally defined transformation unit and rule depending on the semantics of the imported names. More precisely, it associates to each imported name its semantics, to each (name of a) local rule $r$ its corresponding binary relation $\Rightarrow_r$, to each (name of a) local transformation unit $t$ the semantics of $t$ within the environment of $mod$, and the empty set to each other name.

**Definition 5 (semantics of graph transformation modules).** Consider a graph transformation module $mod = \langle IM, DEF, EX \rangle$ over $A$, and a graph transformation environment $E : ID \rightarrow 2^{G \times G}$ defining the imports. Then the module environment of $mod$ with respect to $E$ is the mapping $E_{mod} : ID \rightarrow 2^{G \times G}$ defined by

$$E_{mod}(n) = \begin{cases} E(n) & \text{if } n \in IM, \\ \Rightarrow_r & \text{if } n : r \in DEF, \\ \text{SEM}_{E_{mod}}(\langle I, C, T \rangle) & \text{if } n : \langle I, C, T \rangle \in DEF, \\ \emptyset & \text{otherwise.} \end{cases}$$

The export semantics is obtained as the restriction of the module environment to exported names:

$$EXP_{mod,E}(n) = \begin{cases} E_{mod}(n) & \text{if } n \in EX, \\ \emptyset & \text{otherwise.} \end{cases}$$

Note that the mapping $E_{mod}$ is well-defined because the dependency relation $\sim^M$ is acyclic.

**Remarks**

1. A conventional graph transformation system (consisting of a set $R$ of graph transformation rules) corresponds to a module $mod = \langle \emptyset, DEF, N \rangle$ where $DEF : N \rightarrow R$ defines rules only.

2. A transformation unit $\text{trut} = \langle I, U, R, C, T \rangle$ as in [KK96] corresponds to a graph transformation module $mod = \langle IM, DEF, EX \rangle$ with $IM = U$, $DEF = R \cup \{ t : \langle I, C, T \rangle \}$, and a single export $EX = \{ t \}$.

3. If a graph transformation module $mod = \langle IM, DEF, EX \rangle$ exports all defined transformation units and no rule, every unit $t : \langle I, C, T \rangle$ in $mod$ corresponds to a “[KK96]-transformation unit” $\text{trut}(t) = \langle I, IM \cup U; R; C, T \rangle$ where $R$ are
the rules defined in \( DEF \), and \( U \) are the transformation unit names defined in \( DEF \). Hence, in this case a module corresponds to a set of “[KK96]-
transformation units”.

Notice that all these equivalences only hold up to the handling of names, which is not explicitly treated in the original concept of transformation units.

An example of a module is \( \text{invert\_graph} \) in the upper frame of figure 2. It consists of the rules \( \text{mark} \), \( \text{erase} \), and \( \text{unmark} \). The rule \( \text{erase} \) deletes an unlabeled edge and \( \text{unmark} \) deletes the label \( m \) of an arbitrary \( m \)-labeled edge. The module \( \text{invert\_graph} \) contains also the local transformation unit \( \text{invert} \). The control condition \( \text{mark}!; \ \text{erase}!; \ \text{unmark}! \) requires that first, an \( m \)-labeled edge between every pair of non-adjacent nodes is inserted. Second, all unlabeled original edges are deleted. Third, \( \text{unmark} \) deletes all edge labels \( m \). Hence, the resulting terminal graph of \( \text{invert} \) is the complement of its input graph. The transformation unit \( \text{invert} \) is exported by the module \( \text{invert\_graph} \).

5 Module Systems

A \text{GRACE} program, or \textit{system} is just a (named) collection of graph transformation modules wherein all imports correspond to some export.

**Definition 6.** A named set \( MS \) of graph transformation modules is called a \textit{graph transformation module system}, shortly module system, if the following conditions hold true:

- The names exported by the modules are mutually distinct, i.e. for all modules \( m : \langle IM, DEF, EX \rangle \) and \( m' : \langle IM', DEF', EX' \rangle \) in \( MS \), \( EX \cap EX' \neq \emptyset \) implies \( m = m' \).
- All transformation units or rules imported by a module are exported by another module, i.e. for all \( m : \langle IM, DEF, EX \rangle \) in \( MS \) and all imported names \( n \in IM \), there is a module \( m' : \langle IM', DEF', EX' \rangle \) so that \( n \in EX' \).

The modules composing a \text{GRACE} program are \textit{simple} in that they only restrict the use of names for the programmers. \text{GRACE} programs have a flat semantics, given by the export semantics of the monolithic collection of all definitions of all its modules.

**Definition 7 (flattening of a graph transformation module system).** Let \( S \) be a graph transformation system.

The \textit{flattening} of \( MS \) is the module \( \overline{MS} = \langle \emptyset, \overline{DEF}, \overline{EX} \rangle \) given by the empty import, the disjoint union \( \overline{DEF} \) of all definitions, and the (non-disjoint) union \( \overline{EX} \) of all exports of modules in \( MS \).

The \textit{flat semantics} of \( MS \) is given by the export semantics \( EXP_{\overline{MS}, \emptyset} \) where \( \emptyset \) is the empty environment.
A structured definition of the semantics is based on the additional assumption that the uses relation between modules is well-founded, i.e., descending the chain of imports we are always able to find a module with empty import. Then, the semantics of the modules in the system can be computed bottom-up, starting from these basic modules.

**Definition 8 (uses relation, structured semantics).** Let, in a module system $MS$, the relation $\mathcal{U} \subseteq MS \times MS$ be defined by

$$m : (IM, DEF, EX) \mathcal{U} m' : (IM', DEF', EX') \text{ iff } IM \cap EX' \neq \emptyset$$

Assume that $MS$ is finite, and that the relation $\mathcal{U}$ is acyclic. Then, the semantics of a module $mod$ within $MS$ is defined by $SEM_{MS}(mod) = EXP_{mod,E(mod)}$ where the environment $E(mod)$ is given by

$$E(mod) = \bigcup_{mod \mathcal{U} mod'} SEM_{MS}(mod')$$

Note that the union is disjoint as exports of modules are disjoint. If $mod$ has no import that means $SEM_{MS}(mod) = EXP_{mod,\emptyset}$.

Since we require a hierarchical import structure, the structured and the flattened semantics are equivalent:

**Proposition 1.** For every graph transformation module system $MS$ with acyclic use relation:

$$SEM_{MS}(mod) = EXP_{\overline{MS},\emptyset}$$

**Example 3.** In the following, we define a graph transformation module system that computes maximal independent sets of nodes and cliques in graphs. It implements algorithms given in [Chr75]. The underlying graph transformation approach has been described previously. A set of nodes in a graph is said to be independent if it does not contain two nodes which are connected via an edge. An independent set is maximal independent if the addition of any other node would yield a non-independent set. In contrast with independent sets, a clique is a complete subgraph. To facilitate the computation, we use an interesting relation between those two kinds of sets: cliques of a graph are maximal independent sets of its complement and vice versa.

The module system of figure 2 realizes both algorithms —computing maximal independent sets and cliques— using two modules. The module system is shown as a surrounding box, modules are boxes situated inside.

The semantics of the module `inverted_graph` was already discussed in section 4. The module `sets` imports the transformation unit `invert` from the module `inverted_graph`. It contains the local rule `select` which labels an unlabeled node with $s$ provided that it has no adjacent $s$-labeled node. The local transformation unit `maxiset` applies `select` as long as possible to an unlabeled input graph, i.e. it computes a maximal independent set. Finally, the transformation unit `clique`
**modules:**

invert_graph

- **imports:**
  - *

- **rules:**
  - mark: \( m \rightarrow m \)
  - erase: \( m \rightarrow \)
  - unmark: \( m \rightarrow \)

- **trafos:**
  - invert
    - initial: \( \{ \} \)
    - conds: \( \text{mark}! ; \text{erase}! ; \text{unmark}! \)
    - terminal: \( \text{all} \)
  - exports: invert

- **sets**
  - **imports:** invert
  - **rules:**
    - select: \( \rightarrow \)
  - **trafos:**
    - mariset
      - initial: \( \neg \{ s \} \)
      - conds: select!
      - terminal: all
    - clique
      - initial: \( \{ \} \)
      - conds: invert; mariset
      - terminal: all
  - **exports:** mariset, clique

---

**Fig. 2. A module system**
computes first the complement graph of its unlabeled input graph and determines then a maximal independent set of it. Hence, the \( s \)-labeled nodes of the result of \textit{clique} form a clique of the input graph. Both local transformation units of the module \textit{sets} are exported.

Of course such algorithms can be easily implemented using only transformation units in the sense of [KK96]. But here modules group rules and transformation units according to their use. The first module is graph oriented. It is responsible for inverting a graph. The second module focuses on set theoretic aspects. The transformation unit \textit{invert} which is exported by the module \textit{invert\_graph} is imported and used by the module \textit{sets}. The module \textit{sets} provides two transformation units that can be used by other modules. The export of more than one transformation unit was not possible in \textsc{Grac}, yet.

6 Conclusion

In this paper we have devised a module concept that allows transformations and rules to be encapsulated so that only some of them are exported, and the rest is hidden. The concept is a straight-forward extension of the transformation units defined in [KK96]; it is simple as it just provides \textit{information hiding} for operations, without making the semantics more complicated.

In this section we relate our work to other module concepts for graph transformation in order to identify the issues for further investigations. (See [HEET99] for a more detailed comparison of module concepts for graph transformation.)

Related Work

The module concept recently proposed for \textsc{Progres} [Sch97] is based on the package concept of the object-oriented modeling language UML. Similar to our proposal its emphasis is on information hiding for definitions. However, this includes also the encapsulation graph types which is not (yet) supported in our approach.

In [EE96], distributed states, inheritance, and import-export interfaces are investigated in an approach-independent axiomatic style inspired by modularity concepts for algebraic specifications. However, these concepts remain on an abstract level and are not formally integrated.

The module concept for typed graph transformation systems [CMR96, GPS98] applies the import-export architecture of [EE96] to the DPO approach and uses refinement relations between graph transformation systems in order to model the implementation of exported operations in the body. Also here, graph types may be defined locally, and abstract views of these graphs for other modules can be specified.

DIEGO [TS95] proposes distributed encapsulated graph objects (in the DPO approach) that communicate (and synchronize) via export-import interfaces. As concurrency and distribution is not—yet!—considered for \textsc{Grac}, these concepts do not occur in our proposal too.
Future Work

The main advantage of our approach over the surveyed concepts is that it is completely defined, and still approach-independent. Thus it can be applied to practically every graph transformation approach with only little effort. However, the comparison indicates several aspects by which the presented model has to be extended and refined:

- Functional abstraction (by transformation units) has to be paired with data abstraction for structuring the graphs. If transformation units are tied to such substructures, we get graph classes and objects like in object-oriented languages such as Java [AG95]. (See [Hof99] for a proposal.)
- Given graph objects, concurrency and distribution can be introduced.
- Finally, we can study heterogeneous systems which are composed from modules specified in different graph transformation approaches.

Let's do it soon, and let's do it with grace!

References


[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. Fundamenta Informaticae, XXVII:1/2, 1996.


