Classification and Comparison of Modularity Concepts for Graph Transformation Systems*

Hartmut Ehrig[1], Gregor Engels[2], Reiko Heckel[2], Gabi Taentzer[1]

[1] Technische Universität Berlin, FB 13 Informatik, Franklinstrasse 28/29, D-10587 Berlin, Germany
{ehrig, gabi}@cs.tu-berlin.de
[2] Universität GH Paderborn, FB 17 Informatik, Warburger Str. 100, D-33098 Paderborn
{engels, reiko}@uni-paderborn.de

1 Introduction

In a specification and programming language for state-based systems, structuring concepts are needed both on the level of specifications or programs, and for (the states and computations of) the running systems. In object-oriented approaches, for example, programs are structured into classes while the system at runtime is represented as a collection of objects.

By a *module concept* we mean a structuring concept for specifications or programs (as opposed to run-time structuring) which supports some kind of information hiding. The usual idea is to declare with an export interface which resources (e.g., types and procedures) are provided, and to hide the remaining parts (i.e., their realization), in the encapsulated body of a module. Modules are interconnected by a uses relationship in order to import exported resources of another module. Imported resources are used to realize exported resources of a module.

This paper presents a systematic approach for classifying and comparing modularity concepts that have been proposed for graph transformation systems. The approach is based on the following observation: Modules and module interconnections consist of *basic specifications* (forming, e.g., a module's body, import, or export interface), and *relations* between such specifications, like the implementation relation between export and body of a module. Hence, a natural approach for characterizing a module concept is to answer the following three questions:

1. What are the basic specifications?
2. Which relations between specifications are used?
3. How are specifications and relations combined to modules and interconnections?

In this paper, these questions are considered in some detail for the following module concepts for graph transformation systems.

GCSPEC: Based on the axiomatic notion of *graph class specification*, a module concept inspired by the one for algebraic specifications [EM90] is developed in [EE93,EE96]. In this approach-independent concept, the main structuring techniques are information hiding through interfaces, specialization and refinement, and distribution of states.

GRACE: The GRACE initiative is an attempt for defining a specification and programming language based on graph transformation which is independent of a particular graph transformation approach. The main structuring concept so far are *transformation units* [AEH$^+$96] where information hiding, control conditions, and procedural abstraction are integrated within a single notion. In the recently proposed extension [HHKK98], transformation units are grouped into modules and information hiding is realized on this higher level instead.

DIEGO: DIstributed Encapsulated Graph Objects [TS95] are a structuring concept based on the algebraic DPO approach [CMR$^+$97]. Analogously to classes in OO programming, DIEGO modules provide structuring means both for specifications and at runtime. Thus there are important features which go beyond (what we call) a module concept. Such features, like instantiation of modules and encapsulation of data states, are not sufficiently reflected by the comparison in this paper.

* Research partially supported by the German Research Council (DFG), the TMR network GETGRATS, and the ESPRIT Basic Research Working Group APPLIGRAPH

PROGRES: The first module concept proposed in [WS97b] for PROGRES, a specification and programming language based on PROgrammed Graph REwrite Systems [Sch91] is in the line of classical concepts of programming languages like Modula-2. Our comparison is based on the more recent proposal [WS97a] of a concept inspired by UML packages [Rat97].

TGT: The module concept for typed graph transformation (TGT) in [GPPS98a] uses refinement relations in order to model the implementation of exported features in the body of a module. Analogous to the GCSPEC concept it is inspired by modules of algebraic specifications in [EM90] but provides a concrete instantiation of the abstract ideas of [EE93,EE96].

In addition, the concepts are "tested" w.r.t. a sample modular specification which is presented informally in the next section. As a result of the comparison, at the end we outline some open problems.

2 A Sample Modular Specification

This section introduces the example which shall be used to illustrate and test the various modularity concepts discussed in this paper. It consists of a modular specification of the resource management in a distributed operating system (derived from [Hec98]). Figure 1 shows the structure of the overall specification where boxes represent modules and arrows model *uses* relationships.

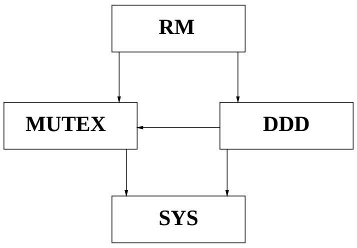
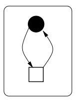


Fig. 1. The module structure.

The the top-level module RM integrates the functionalities of the modules $MUTEX$ and DDD which realize, respectively, mutual exclusion and distributed deadlock detection. They are, in turn, based on the module SYS that provides the basic system types and operations for processes and resources. We present the modular specification in a bottom-up way, starting with the export interface of module SYS.

SYS =
export
types



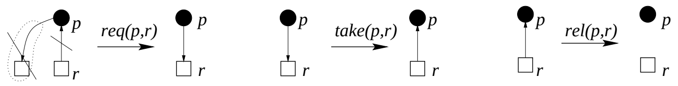
procedures

The type graph may be read like an entity/relationship schema specifying the node and edge types which may occur in instance graphs modeling system states. Processes P are drawn as black nodes and resources R as light boxes. An edge from a process to a resource models a *request*. An edge in the opposite direction shows that the resource is currently *held_by* the process. Besides, procedures are provided for creating and deleting processes, and for mounting and unmounting resources. The above rules are just partial descriptions of the corresponding operations which have to be fully implemented in the hidden body of the module.

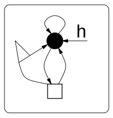
Module *MUTEX* implements a mutual exclusion algorithm. It exports procedures which allow processes to issue a request for a resource, take it over, and releasing the resource upon completion of their task. The negative application conditions [HHT96] for $req(p, r)$ ensure that each process issues only one request at a time. The *public* import of module *SYS* within the export interface indicates that the types and operations of *SYS* are also available to the users of *MUTEX*.

In the body, the mutual exclusion is implemented by means of a token ring algorithm, i.e., a cyclic list of processes, where an edge between two processes points to the *next* process. A pointer marks the *head* of the list as default position for introducing new processes and resources. For each resource there is a *token*, represented by an edge with a white flag, which is passed from process to process along the ring. If a process wants to use a resource, it waits for the corresponding token. Mutual exclusion is ensured because there is only one token for each resource in the system.

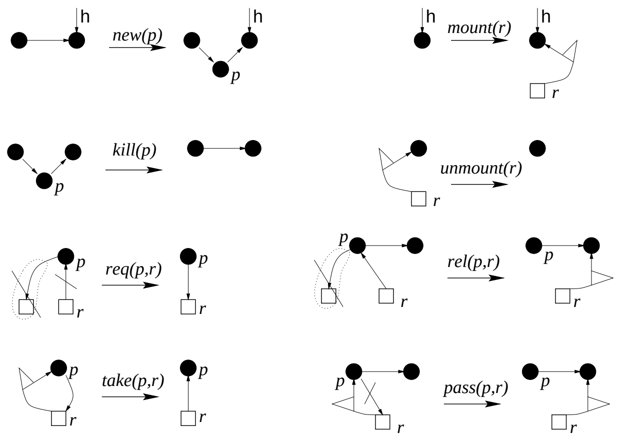
MUTEX =
export
from *SYS* **import** *all*
procedures



body
types



procedures

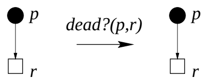


The type graph of the *MUTEX* body extends the imported types of *SYS* by edge types *next*, *head*, and *token*. The rules are extended correspondingly in order to take care of the ring structure and the token. Notice how this allows to hide the token ring implementation while providing a graphical interface to the *MUTEX* module. In particular, the *SYS* operations are exported like they are imported, but internally they are extended for dealing with the additional structure of the token ring.

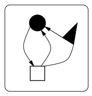
Module *DDD* implements a distributed deadlock detection algorithm [CMH83] which may be invoked with the $dead?(p, r)$ rule by a process waiting for a resource.

In a graph representing a state, a deadlock is represented as a cycle of *request* and *held_by* edges. In order to check for a deadlock cycle, a *blocked*-message created by $dead?(p, r)$ is passed around using $blocking(p, r)$ as long as the resource r held by the receiver of the message is not the original resource. This is ensured by the inequation, which is also a negative application condition in the sense of [HHT96]. If the receiver does not hold any resource, it deletes the message with $ignore(p, r)$. Thanks to the mutual exclusion, each resource is held by only one processes. Hence, if the message arrives at a process who holds the original resource, this is the original sender of the message. The deadlock thus detected is broken by the $rel(p, r)$ rule which is imported from the $MUTEX$ module.

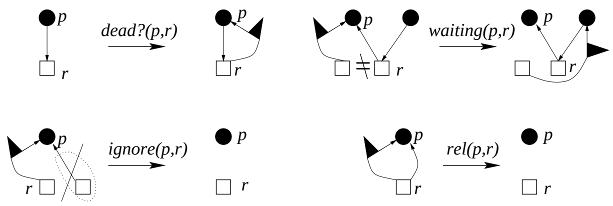
DDD =
  export
    from SYS **import** *types*
    procedures



  body
    from $MUTEX$ **import** $rel(p, r)$
    types



    procedures



Finally, mutual exclusion and deadlock detection are combined in a single module RM which provides a unique interface to the operations defined above. In particular the scenario of a process requesting a resource, invoking several times the deadlock detection if access is not granted in time and finally taking it is encapsulated in the procedure $open(p, r)$. The closing procedure instead is just a renaming of $rel(p, r)$.

RM =
  export
    types P, R
    procedures
      $new(p : P), kill(p : P)$
      $mount(r : R), umount(r : R)$
      $open(p : P, r : R), close(p : P, r : R)$
  body
    from $MUTEX$, DDD **import** *all*
    procedures
      $open(p, r) = req(p, r); dead?(p, r)^*; take(p, r)$
      $close(p, r) = rel(p, r)$

3 What are the basic specifications?

A *graph transformation specification* consists of a set of rewrite rules and additional structuring means *in-the-small*. In particular, we consider *graph types*, given e.g., by a graph schema (or type graph) and *graph procedures* specified either by simple rules or by transactions (PROGRES) respectively transformation units (GRACE) integrating aspects of procedural abstraction and control.

The components of the modules in the previous section provide various examples of basic specifications. They are largely approach independent but for the fact that not all specification concepts are supported in all the approaches: Procedures with formal parameters (which are standard in every programming language) are explicitly supported only in PROGRES. Control structures like in the body of *RM* are available in GRACE and PROGRES while in TGT refinement concepts serve a similar purpose.[1] The corresponding features are summarized in Table 1 where 'x' indicates full and '(x)' partial support (i.e., GCSPEC does not know control structures and GRACE no formal parameters).

Semantically we could distinguish between sequential and concurrent semantics as well as between classical and loose semantics. In particular, the loose semantics for graph productions turned out to be very useful for compositional reasoning about graph transformation systems which are composed of several views [EHTE97,Hec98]. However, since the *concrete*[2] module concepts all come with a classical, sequential semantics, we do not consider this aspect.

	types	rules	procedures
GCSPEC	x	x	(x)
GRACE	x	x	(x)
DIEGO	x	x	
PROGRES	x	x	x
TGT	x	x	

Table 1. Graph transformation specifications: structuring in-the-small.

4 Which relations between specifications are used?

Modules consist of basic specifications and relations between them. Also module interconnections, like the uses relation and the composition of modules, are implemented by means of relations (and operations) on specifications. Table 2 surveys the relations used in the various module concepts. Conceptually, we distinguish three major aspects of such relations, namely implementation, specialization, and refinement. Often, the relations used in the concepts combine several of these aspects.

A relation like the one between export and body of the *RM* module in Section 2 is called *implementation*. Here the export does not contain semantical information but only declarations of types and procedures. In the GRACE module concept, for example, the export interface is just a set of names of transformation units and rules, and the body provides for each name the actual definition. A similar observation can be made for PROGRES, even if there it is possible to specify interface rules and procedures as comments for the user of the module.

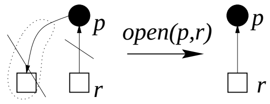
This idea is generalized by a *specialization* which allows to *extend an existing implementation* given by types, rules, and graph procedures. The semantical idea is a projection of states and transformations from the target to the source of the specialization relation. Rules and procedures in the specialized system may have additional effects and stronger application conditions. Notice that there is a trade-off between encapsulation and graphical specification (cf. also [WS97b]). Classically, the export interface only shows the signatures of procedures while hiding completely

[1] Of course, independent from module concepts, corresponding notions exist also in other approaches, see e.g., [HMTW95,HEWC97,Hec98] for extensions of the algebraic approach by control structures and/or parameterization.

[2] In the *generic* concepts GCSPEC and GRACE a loose semantics could be assumed in the definition of the graph transformation approach.

the implementation. In this case, however, imported procedures or rules can only be invoked within (mostly textual) control conditions like in the body of *RM*. On the other hand, making available part of the implementation like in the case of the *MUTEX* module allows, e.g., the *DDD* body to extend the imported $rel(p, r)$ rule by the additional effect of deleting the blocked message (represented by the black flag). Notice that still the implementation of *MUTEX* by the token ring is hidden.

A *refinement relation* maps an elementary operation of the more abstract specification to a composite operation in the more concrete specification. For typed graph transformation systems, temporal and spatial refinement relations have been defined in [GPPS98b]. In a spatial refinement, each rule is refined by an amalgamation (i.e., a parallel composition with sharing) of rules, while in a temporal refinement it is refined by a sequential composition. In the TGT concept [GPPS98a] refinement relations are used for modeling implementations. This is needed in particular for concepts that do not provide explicit means for procedural abstraction. For example, the implementation of the $open(p, r)$ operation in the *RM* module could be modeled in the TGT concept by a temporal refinement of the exported rule



by the sequential composition $req(p, r); dead?(p, r)^n; take(p, r)$ which, when evaluated for any finite number n of occurrences of the (idle) $dead?(p, r)$ rule, yields exactly the original rule $open(p, r)$.

	implementation	special. of types	special. of rules	refinement of rules
GCSPEC		x	x	
GRACE	x			
DIEGO			x	
PROGRES	x	x		
TGT		x	x	x

Table 2. Aspects of relations between basic specifications.

5 How are specifications and relations combined to modules and interconnections?

Modules in programming and specification languages use a variety of different interfaces, e.g., *export interfaces* for restricting the use of resources and *import interfaces* expressing requirements for imported resources. In OO approaches the export is further distinguished into public, private, and protected parts. Thus, a major characteristics of each module concept is the *form* of its modules.

We use a diagrammatic notation in order to discuss this aspect. Consider, as a simple example, the form of a PROGRES package given by the *body* B and the *public interface* E (for export).[3] The relation $B \leftarrow E$ is an implementation *impl*.

In a similar way we describe relations and operations on modules. In PROGRES packages, for example, two kinds of import are distinguished, called *public* and *protected* import. The difference is that a resource which is imported public, is included in the public (i.e., export) interface E and is thus made available also to its clients. A protected import is included instead in the body B. In

[3] For simplifying the presentation we use a uniform notation for the module components in the various concepts.

our sample specification, module *SYS* is imported public by *MUTEX* and *DDD* while all other imports are protected.

$$\begin{tikzcd} B & E \arrow[l, "imp"'] \end{tikzcd} \qquad \begin{tikzcd} B & E \arrow[l, "imp"'] \\ B' & E' \arrow[l, "imp"'] \arrow[u, hook] \end{tikzcd} \qquad \begin{tikzcd} & B & E \arrow[l, "imp"'] \\ B' & E' \arrow[l, "imp"'] \arrow[u, hook] & \end{tikzcd}$$

PROGRES package *public import* *protected import*

The GRACE module concept, also based on implementation relations, is additionally provided with an interface for abstract import serving as a formal parameter for the actually imported resources: This concept supports a top-down development of a module system by anticipating in the import interface the functionalities of modules yet to be implemented. In case of an abstract import interface, the use of several modules is realized by a *union* (cf. second-last column in the Table 3): The import interface I of a client module is included in the export interface $\bar{E}$ of the module $\bar{I} \leftarrow \bar{B} \rightarrow \bar{E}$ obtained as the (partly disjoint) union of all modules from which resources shall be imported.

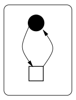
$$\begin{tikzcd} I \arrow[r, "imp"] & B & E \arrow[l, "imp"'] \end{tikzcd} \qquad \begin{tikzcd} & & I \arrow[r, "imp"] \arrow[d, hook] & B & E \arrow[l, "imp"'] \\ \bar{I} \arrow[r, "imp"] & \bar{B} & \bar{E} \arrow[l, "imp"'] & & \end{tikzcd}$$

GRACE module *uses relation*

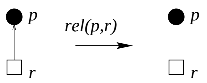
For the modules of our sample specification no abstract import is used (thus the bottom-up presentation). It can be defined by collecting the imported types and procedures from the export and body parts of the importing module in an extra import interface. In case of module *DDD*, for example, the following interface is constructed.

DDD import

types



procedures



Now, the use of *MUTEX* and *SYS* is realized by mapping this import interface to the *MUTEX* export (which in this case already contains the *SYS* types, so the union of *MUTEX* and *SYS* is not needed for realizing the multiple import. However, in this way, the distinction between public and protected import is lost.

The refinement-based module concept (TGT) has almost the same form of modules except for the relations that are used. Here, specializations *spc* connect the import to the body and refinements ref relate the export with the body. Moreover, operations of module composition and union are supported, which allows to build complex module systems while hiding internal uses relationships.

$$
\begin{tikzcd}
 & E \arrow[d, "ref"'] \\
I \arrow[r, "spc"] & B
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
 & & E \arrow[d, "ref"'] \\
 & I \arrow[r, "spc"] \arrow[d, "spc"'] & B \arrow[dd, dotted, "ref"] \\
 & E' \arrow[d, "ref"] & \\
I' \arrow[r, "spc"] & B' \arrow[r, dotted, "spc"] & B''
\end{tikzcd}
$$

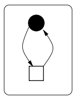
TGT module *composition*

In the case of the module system in Section 2, this would allow to derive a single module with the export interface of *RM* and the import of *SYS* by composing (in this order) *SYS, MUTEX, DDD*, and *RM*.

GCSPEC and DIEGO modules provide, in addition to body, import, and export, a formal parameter interface P that allows, in the presence of an abstract import interface, for the distinction between public and protected import discussed above (see diagram below on the left). In fact, in our examples, P would contain those resources which are imported within the export interface. For the *DDD* module the parameter part is shown on the right below.

$$
\begin{tikzcd}
P \arrow[r] \arrow[d] & E \arrow[d] \\
I \arrow[r] & B
\end{tikzcd}
$$

DDD parameter
types



The relations between the component specifications are the ones discussed in the previous section.

The features of the various concepts are summarized in Table 3. The first three entries of each line refer to the form of modules as discussed above. We speak of a modular semantics if a module is not just syntax but provides a semantical mapping from import to export semantics. This requires an abstract import and some form of loose semantics. The last three entries list the interconnection mechanisms defined (or discussed) in the approaches.

	export	abstract imp.	public imp.	mod. sem.	use rel.	union	comp.
GCSPEC	x	x	x	x	(x)	(x)	(x)
GRACE	x	x		x	x	x	
DIEGO	x	x	x		x		
PROGRES	x		x		x		
TGT	x	x			x	x	x

Table 3. Modules: Interfaces, semantics, and interconnection mechanisms.

6 Conclusion

This paper provides an informal approach for describing, comparing, and classifying modularity concepts in the context of graph transformation together with an example of a modular specification for illustrating the main ideas. None of the so far published approaches supports all the specification concepts that were used in the example while all aspects where at least supported by some of the approaches. Thus, a major theme of the future research is the integration of the various aspects into a single concept.

In particular, future topics are the refinement and specialization of complex graph procedures and the incorporation of loose semantics and concurrency on the basis of [Hec98]. Also, runtime structuring features like encapsulation of data states and instantiation are highly desirable but are beyond the scope of this paper. This is partly due to the lack of available concepts for comparison (DIEGO being the only one). But it is also an open question how, for example, distribution and specialization are related.

Finally it shall be noted that, since this is not a formal paper, much of the classification here is based on intuition and interpretation, and of course open for discussion (which is in fact the actual purpose of such a paper). Moreover, most of the surveyed concepts are still in a preliminary state and may be subject to rapid changes. Nevertheless, we are confident that (in particular at this stage) a discussion and clarification of concepts is helpful for developing new proposals and combining existing ones.

Acknowledgment The authors wish to thank Hans-Jörg Kreowski for his detailed comments on a preliminary version of this paper.

References

[AEH+96] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 1998. To appear. Also as TR 7/96, University of Bremen, 1996.

[CMH83] K.M. Chandy, J. Misra, and L.M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1:144–156, May 1983.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*. World Scientific, 1997.

[EE93] H. Ehrig and G. Engels. Towards a module concept for graph transformation systems. Technical Report 93-34, Leiden University (The Netherlands), 1993.

[EE96] H. Ehrig and G. Engels. Pragmatic and semantic aspects of a module concept for graph transformation systems. In *LNCS 1073 , Proc. Williamsburg, U.S.A.*, pages 137–154. Springer Verlag, 1996.

[EHTE97] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A combined reference model- and view-based approach to system specification. *Int. Journal of Software and Knowledge Engeneering*, 7(4):457–477, 1997.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1990.

[GPPS98a] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinements and modules for typed graph transformation systems. Technical Report SI 98/05, Universitá di Roma *La Sapienza*, Dip. Scienze dell'Informazione, 1998.

[GPPS98b] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Spatial and temporal refinement of typed graph transformation systems. In *Proc. of MFCS'98, Brno*, 1998. to appear.

[Hec98] R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems*. PhD thesis, TU Berlin, 1998.

[HEWC97] R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Loose semantics and constraints for graph transformation systems. Technical Report 97-07, TU Berlin, 1997. http://www.cs.tu-berlin.de/cs/ifb/TechnBerichteListe.html.

[HHKK98] R. Heckel, B. Hoffmann, P. Knirsch, S. Kuske. Simple Modules for GRACE. In this volume.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.

[HMTW95] R. Heckel, J. Müller, G. Taentzer, and A. Wagner. Attributed graph transformations with controlled application of rules. In G. Valiente and F. Rossello Llompart, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science*. Technical Report B - 19, Universitat de les Illes Balears, 1995.

[Rat97] Rational software cooperation. Unified modeling language. available via *http://www.rational.com*, 1997.

[Sch91] A. Schürr. Progress: A vhl-language based on graph grammars. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 532*. Springer, 1991.

[TS95] G. Taentzer and A. Schürr. DIEGO, another step towards a module concept for graph transformation systems. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[WS97a] A.J. Winter and A. Schuerr. Adopt, adapt, and improve: Uml packages for graph rewrite systems. In *1st GETGRATS Workshop, Bordeaux*, October 1997. See also the corresponding paper in this volume.

[WS97b] A.J. Winter and A. Schuerr. Modules and updatable graph views for programmed graph rewriting systems. Technical Report 97-3, RWTH Aachen, FG Informatik, October 1997.