Chapter 1

CLASSIFICATION AND COMPARISON OF MODULE CONCEPTS FOR GRAPH TRANSFORMATION SYSTEMS

R. HECKEL, G. ENGELS
Universität GH Paderborn,
FB 17 Informatik, Warburger Str. 100, D-33098 Paderborn
{reiko, engels}@uni-paderborn.de

H. EHRIG, G. TAENTZER
Technische Universität Berlin,
FB 13 Informatik, Franklinstrasse 28/29, D-10587 Berlin, Germany
{ehrig, gabi}@cs.tu-berlin.de

In this chapter we use a three-level approach for analyzing module concepts for graph transformation systems. Based on the observation that module architectures consist of *basic specifications* like body or export interface and *relations* like the implementation relation between them, or the import relation between a client and a server module, module concepts are characterized by answering the questions

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

These three questions are investigated for five module concepts available in the literature. As a reference example, a sample modular specification of resource management in distributed operating systems is presented.

1.1 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. Resources exported by other modules are used to implement the own export.
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: Module architectures (i.e., modules and their 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 inside a module or the import relation between client and server modules. 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 module architectures?

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

GCSPEC: In [2,3] different structuring concepts like distributes states, inheritance, and import-export interfaces are presented in a conceptual framework for modularity of graph transformation systems. The presentation is based on the notion of *graph class specification* consisting of graph types, procedure symbols, and constraints. This is understood in analogy to sorts, operation symbols, and equations in algebraic specifications [4]. In particular, the import-export interface concept is inspired by the notion of algebraic specification module [5]. In contrast to other approaches surveyed in this chapter, GCSPEC does not provide a concrete implementation of a module concept but states requirements for such concepts on an axiomatic bases.

[a] Some familiarity is assumed with the corresponding graph transformation approaches, see [1] for a recent survey.

GRACE: As an attempt for defining a common graph-centered specification and programming language, GRACE [6,7] is based on an axiomatic definition of *graph transformation approach* which covers the basic notions of most approaches in the literature. Elementary structuring means in GRACE are *graph transformation units* [7] which integrate some aspects of information hiding, control, and procedural abstraction. Recently, a simple notion of module has been proposed [8] in order to group together several transformation units and lift the task of information hiding to this higher level.

DIEGO: DIstributed Encapsulated Graph Objects [9] are a structuring concept based on the algebraic DPO approach [10]. 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.

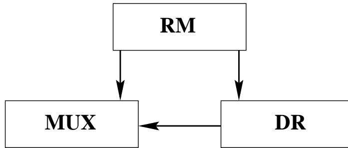
PROGRES: PROgrammed Graph REwrite Systems [11,12] provide a specification and programming language based on graph transformation which has been developed as part of the IPSEN project [13] for the specification and prototyping of integrated software engineering environments. The first module concept for PROGRES proposed in [14] is in the line of classical concepts of programming languages like Modula-2. Our comparison is based on the more recent proposal [15] of a concept inspired by the package concept of UML [16].

TGTS: Typed Graph Transformation Systems [17,18] combine the idea of a graph schema or type graph for specifying the structure of states with the concept of graph transformation for modeling the dynamics. In [19] a module concept for typed graph transformation systems in the DPO approach [10] is proposed. It employs a similar module architectures like algebraic specification modules [5] and uses refinement relations between graph transformation systems [20,21] in order to model the implementation of exported features in the body.

As a reference example for these concepts, a sample modular specification is presented informally in the next section. Then, the concepts are analyzed w.r.t. questions 1 to 3. In the end, the results are summarized and related open problems are discussed.

1.2 A Sample Modular Specification

This section introduces the example which shall be used to illustrate and analyze the module concepts discussed in this paper. It consists of a modular specification of the resource management in a distributed operating system. The figure below shows the architecture of the overall specification where boxes represent modules and arrows *import* relationships. The top-level module *RM* for Resource Management integrates the functionalities of the modules *MUX* and *DR* which realize, respectively, MUtual eXclusion and Deadlock Recovery. Moreover, *DR* imports from *MUX* some types for processes and resources as well as a *release* operation for resolving detected deadlocks.



The presentation is not tailored towards a particular approach, even if the example is derived from a small case study in the PhD thesis of the first author [22] in the DPO approach. An *ad-hoc* syntax is used for module concepts: A module consists of two parts, the **export** and the **body**. In the export **types** and **procedures**[b] may be declared or included as **import** from other modules. In the body we have to **implement** the declared resources possibly using additional imports. An import in the export part is called *public import* since all imported resources are also available to clients. An import in the body is referred to as *hidden import*.

The classical form of import, called *direct import*, specifies a server module and lists the imported resources. Instead, in a **generic** import the server module is left open, that is, the listed resources are just formal templates or place-holders which have to be replaced (instantiated) by actual resources. Constraints can be specified on these templates as properties to be fulfilled by the actual resources.

The module *MUX* in Figure 1.1 uses a generic import in order to be independent of the underlying operating system. In the export of *MUX*, for example, the generic import of basic system types can be found. They are shown as a *type graph* which may be read like an Entity-Relationship schema specifying

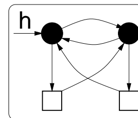
[b]In this chapter, a procedure is often just a graph transformation rule equipped with a name and a list of formal parameters in order to support procedural abstraction. In general, this concept also subsumes control structures and recursion.

the node and edge types that 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. Thus, in addition to an enumeration of types, the type graph specifies constraints for the connection of node and edge types which have to be verified w.r.t. the types of a concrete server module when instantiated.

Analogously, the rules of the generic import in the body of MUX are interpreted as constraints for the actual operations for creating and killing processes, and mounting and unmounting resources.[c] E.g., in the case of *new(p)*, it is assumed that by the actual operation *at least* a process p is created, but additional unspecified effects may take place, for example, in order to update the system's internal data structures. Thus, such interface rule is not intended as implementation of an operation but specifies a lower bound to its effects. This idea is formalized in [23] by a loose semantics of graph transformation rules.

Beside the imported types and the system procedures (which are re-exported with the same names and constraints), the export of module MUX declares the procedures *req, take*, and *rel* which form the actual interface to the mutual exclusion algorithm. They allow processes to issue requests, take resources, and release them upon completion of their task. The negative application conditions [24] for $req(p, r)$ ensure that each process may issue only one request at a time.

In the body, the mutual exclusion is implemented by means of a token ring algorithm. The token ring is 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 free 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 at most one token for each resource in the system. The graph below represents a sample state of two resources and two processes each holding one resource while requesting another one.



The relations between generic import and body could be formally described as

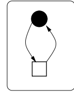
[c] Since these operations are inverse to each other we use an abbreviating "back-and-forth" notation.

MUX =

export

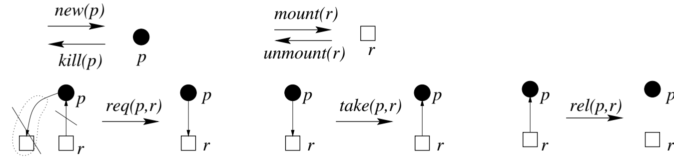
import generic

types



declare

procedures



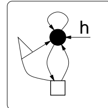
body

import generic

procedures

implement

types



procedures

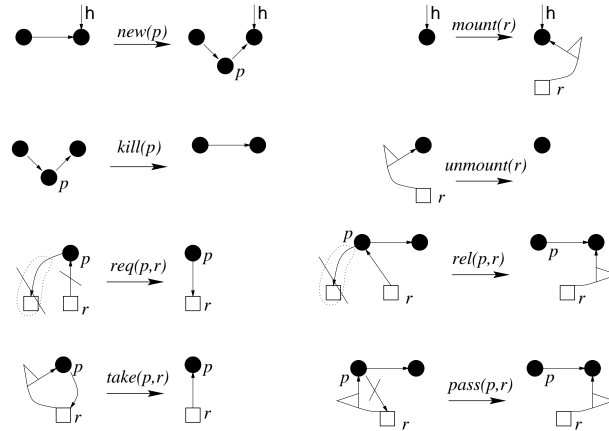


Figure 1.1: Module MUX.

view relations in the sense of [23]. The type graph of the *MUX* body extends the imported types by edge types *next*, *head*, and *token* (shown, respectively, as loop, *h*-labeled edge, and flag). The imported rules are extended in the body in order to take care of the ring structure and the token. A similar relation is defined between export and body hiding again the token ring implementation. This leads to the situation that the system operations are exported like they are imported, but internally they are extended for dealing with the additional structure of the token ring.

Module *DR* in Figure 1.2 implements an algorithm for (distributed) deadlock recovery [25] which may be invoked with the $rec_dl(p,r)$ 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. This is the case, for example, in the sample state above which shows a deadlock of two processes and two resources. In order to check for a deadlock cycle, a *blocked*-message (the black flag created by $rec_dl(p,r)$) is passed around using $waiting(p,r)$ 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 [24]. 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 process. Hence, if the message arrives at a process which 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 *MUX* module.

In order to understand the interaction between a client and a server module let's consider the rule $rel(p,r)$ which is imported from *MUX* in the body of *DR*. The only effect which is specified for this procedure by the interface rule in the export of *MUX* is the deletion of the *held_by* edge from the resource to the process. The bodies of *MUX* and *DR* provide two different extensions of this rule which may be seen as two different views (in the sense of [23]) of the same overall operation. Beside the deletion of the *held_by* edge which is common to both views, in the *MUX* view of the procedure the token is created and passed to the next process. In the *DR* view, this rule is used in order to resolve a deadlock. The additional action consists in the deletion of the *blocked* message. The overall interaction of a server and client module can be realized by gluing the bodies of the two modules over the common interface using the construction of view integration in [23].

Mutual exclusion and deadlock recovery are encapsulated (by hidden import) in a single module *RM* in the lower part of Figure 1.2 which provides a unique interface to the operations implemented by *MUX* and *DR*. In particular, the scenario of a process requesting a resource, invoking the deadlock recovery until access is granted, and finally taking it over, is integrated in the procedure

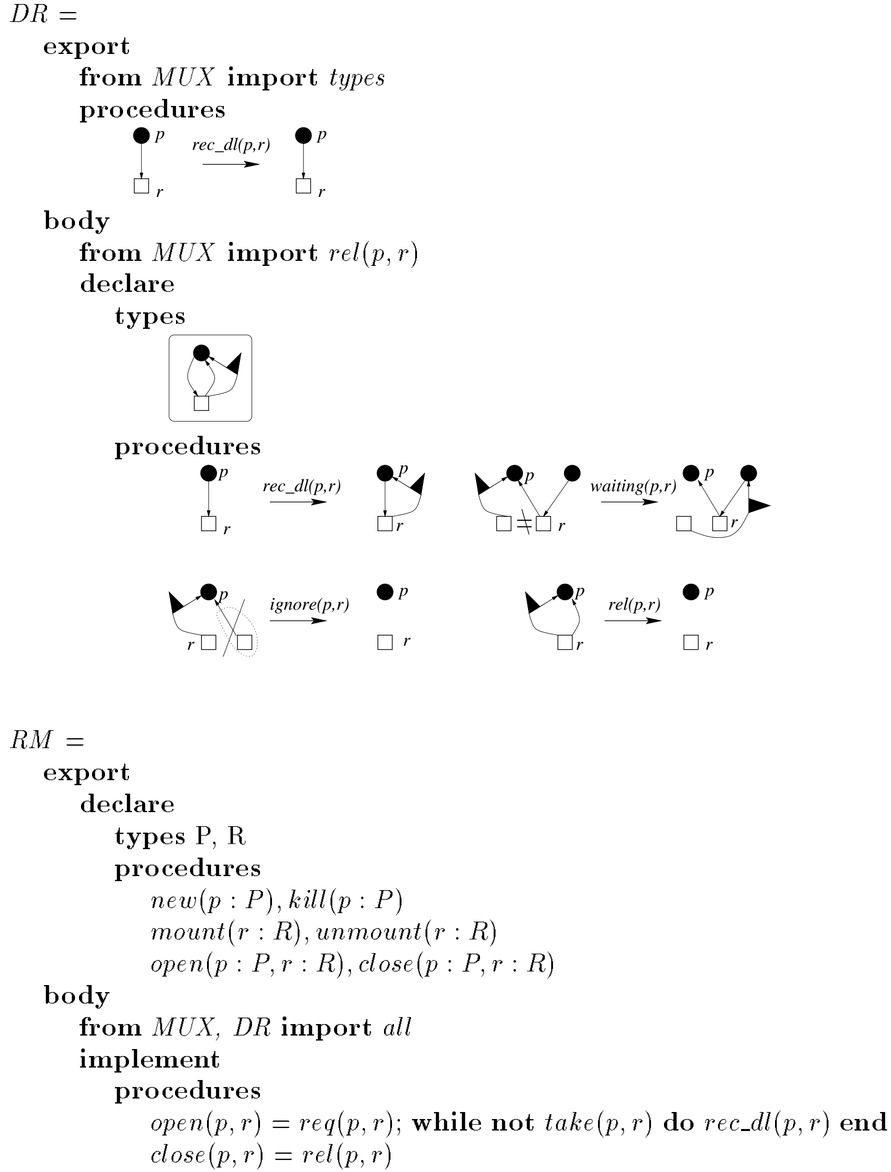
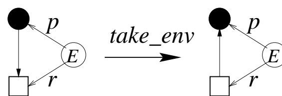


Figure 1.2: Modules *DR* and *RM*.

$open(p, r)$. The closing procedure instead is just a renaming of $rel(p, r)$. Notice that we use a purely textual presentation. The types P and R are symbolic representations of the process and resource types shown in the type graph exported by MUX as black circle and light box, respectively.
In the following three sections this example is used in order to analyze graph transformation module concepts w.r.t. their expressiveness. Following the outline of the three questions listed in the Introduction we discuss the support for structuring features and sketch possible alternatives. The first question is

1.3 What are the basic specifications?

That is, which specification concepts *in-the-small* may be used, for example, in the body of a module? In general, a *(basic) graph transformation specification* (in whatever approach) consists of a *graph type* declaration, given e.g., by a type graph or by a set of labels, and a set of *graph procedures*. They may be specified either by simple rules or by transactions (PROGRES) respectively transformation units (GRACE) which integrate aspects of procedural abstraction and control.
The components of the modules in the previous section provide various examples of basic specifications. Not all the specification concepts that are used are supported in all the graph transformation 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 GCSPEC they are subsumed by the more general concept of constraints. In TGTS refinement concepts serve a similar purpose. The corresponding features are summarized in Table 1.1 where 'x' indicates full and '(x)' partial support (i.e., GCSPEC and GRACE provide graph procedures but no mechanism for parameter passing).
The last column of the table surveys the semantics of basic specifications: GCSPEC, GRACE, and PROGRES use relations on graphs, obtained in the last two cases through a fix-point (FP) construction [26,27]. In DIEGO and TGTS the semantics of basic specifications is given in terms of DPO derivations.
The absence of concepts like procedural abstraction or control in one of the approaches does not necessarily mean that the reference example cannot be modeled in this approach. The parameter passing mechanism for procedures, for example, can be implemented to some extend by encoding an environment into the graphical structure as shown below for the rule $take(p, r)$.



	types	rules	procedures	control	semantics
GCSPEC	x	x	(x)	x	relations
GRACE	x	x	(x)	x	relations (FP)
DIEGO	x	x			derivations
PROGRES	x	x	x	x	relations (FP)
TGTS	x	x			derivations

Table 1.1: Graph transformation specifications: structuring in-the-small.

Here the environment is represented by the new node labeled with E and its outgoing edges labeled by p and r. A call to $take(p, r)$ is implemented by inserting the appropriate pointers to the actual parameters (i.e., the concrete process and resource nodes to which the rule shall be applied) and by invoking the modified rule $take_env$. Similar encodings are possible for simple control structures. However, in our view such encodings lead to a bad specification style since program structures are mixed up with data structures. In general, specification concepts in-the-small lead to more structured and readable specifications.
Modules are built on top of these elementary concepts in order to group together related types and procedures and to restrict their visibility. Next we discuss the relations that may be used to relate two different modules or two components within the same module.

1.4 Which relations between specifications are used?

Modules consist of basic specifications and relations between them. Also module interconnections, like the import relation and the composition of modules, are implemented by means of relations and operations on specifications. Table 1.2 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. Since the semantical interpretation is different in each case, it is not listed in the table but explained in the course of discussion.
A relation like the one between export and body of the module RM in Section 1.2 is called *implementation*. Here the export does not provide semantical information but only types and signatures of 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. Since the body may contain auxiliary procedures which are not declared in the export, the implementation relation corresponds to a restriction

of the relations associated with the body procedures to the exported ones.

The first paper [14] on modularity in PROGRES noticed a trade-off between encapsulation and graphical specification. Classically, the export interface of a module only shows the signatures of procedures while hiding completely the implementation. In this case, however, imported procedures can only be used within (mostly textual) control structures like in the body of *RM*. On the other hand, making available part of the implementation like in the case of the *MUX* module in Figure 1.1 allows, e.g., the *DR* body in Figure 1.2 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 *MUX* by the token ring is hidden.

This more general relation which allows to extend an existing implementation given by types, rules, and graph procedures is called *specialization*. The semantical idea is a projection of states and transformations from the target to the source of the relation. Hence, specialized rules and procedures may have additional effects and stronger application conditions. This is implemented, for example, by the notion of view relation in [23].

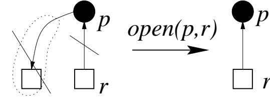
In the package concept for PROGRES [15] the idea of specialization is present by a flexible visibility concept which distinguishes between *public, protected*, and *private* items. The public items provide an interface which supports full encapsulation and which is related to the body of a module by an implementation relation. The protected part may reveal some aspects of the implementation and is thus connected by a specialization. The implementation of procedures, however, is usually private since specialization of complex control structures like they are used in PROGRES is still an unsolved problem. The trade-off between encapsulation and graphical specification is resolved in favor of the latter: A client package A can use its own rules to modify data structures of another packages B provided the corresponding types are marked as *mutable*. From the point of view of the exporting package B these changes are not specified. Conceptually, this means that PROGRES packages may have a loose semantics with mutable types corresponding to open types in the sense of [23]. Inconsistencies which may arise from this kind of interaction between client and server packages are resolved in [15] by active constraints, i.e., special graph procedures which test for the violation of certain integrity constraints and start appropriate repair actions.

A *refinement* maps an elementary operation of the more abstract specification to a composite operation (with the same effect) in the more concrete specification. For typed graph transformation systems, temporal and spatial refinement relations have been defined in [20]. In a spatial refinement, each rule is refined by an amalgamation (i.e., a parallel composition with sharing) of rules, while in

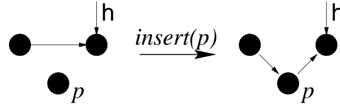
	implementation	specialization of types	specialization of rules	refinement of rules
GCSPEC		x	x	
GRACE	x			
DIEGO			x	
PROGRES	x	x		
TGTS		x	x	x

Table 1.2: Aspects of relations between basic specifications.

a temporal refinement it is refined by a sequential composition. Both kinds of refinement are combined in [21]. In the TGTS module concept [19] refinement relations are used for modeling implementations. This is needed in particular for concepts that do not provide explicit means for procedural abstraction and control. For example, the implementation of the $open(p, r)$ operation in the RM module could be modeled in the TGTS concept by a temporal refinement of the exported rule



by the sequential composition $req(p, r); rec_dl(p, r)^n; take(p, r)$ which, when evaluated for any finite number n of occurrences of the (idle) $rec_dl(p, r)$ rule, yields exactly the original rule $open(p, r)$. Semantically, a refinement means that a single step in the source system may be replaced by a sequence or parallel composition of steps in the target system.
Refinements or control structures can also be used to realize the specialization of rules. The rule $new(p)$ in the import of MUX in Figure 1.1, for example, is extended in the body in order to insert the newly created process in the token ring. The same effect can be obtained by the procedure $new(p) := import.new(p); insert(p)$ where $import.new$ refers to the imported rule and $insert(p)$ is given by



Thus, applying $new(p)$ is conceptually equivalent to applying first the imported

rule $import.new(p)$ and then the rule $insert(p)$ implementing the additional effects.

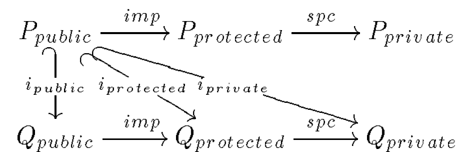
After having discussed basic specifications as well as possible ways to relate them, in the next section we adopt the more abstract view asking for the architecture of modules and their interconnections.

1.5 How are specifications and relations combined to module architectures?

The modules in our reference example use a variety of different interfaces for export, hidden or public and direct or generic import. Since most of the discussed module concepts support only some of these interfaces, a major characteristics of each concept is the *form* of its modules.

We use a diagrammatic notation in order to discuss this aspect. Consider, as an example, the form of a PROGRES package P in the diagram below. Such package consists of a *public* part (corresponding to the export interface) a *private* part (corresponding to the body) and an intermediate *protected* part which contains the public part and is embedded in the private part. Public and protected part are related by an implementation *imp* and a specialization *spc* embeds protected into private.

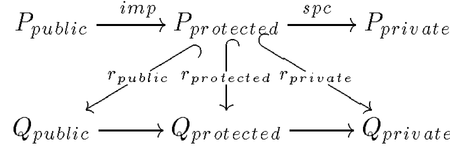
In a similar way we describe relations and operations on modules. In PROGRES packages, six kinds of relations between modules are distinguished, namely *public, protected*, and *private import* as well as the three corresponding notions of *refinement*.[d] In general, one speaks of import if the public part of the server module is included in the client while refinement means that the protected part is included. The attributes *public, protected*, or *private* specify in which part of the client module the resources are included. The three kinds of import are depicted below where P is the server and Q the client package.



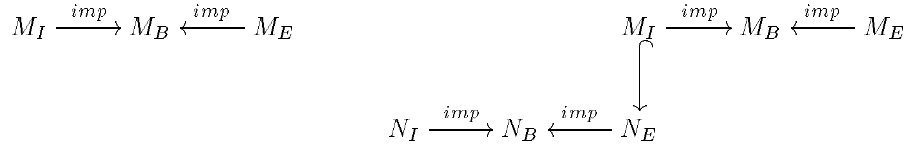
The arrows labeled by i_x show the type of inclusion for the import variant x. Using the same notation the three variants of refinement r_x are shown in the

[d] In our terminology, the refinement of [15] is a specialization relation.

next diagram.



In our reference example we didn't distinguish between public and protected resources. Hence of the six combinations only two are possible, called public and hidden import. However, the export of *MUX*, for example, plays the same conceptual role like a protected part in [15]. The public import of *MUX* in *DR* can therefore be seen as a public refinement.
The GRACE module concept is additionally provided with an interface for generic import which serves as a formal parameter for the actually imported resources. In our sample specification, a generic import is used in order to keep the *MUX* module independent of the underlying system. This concept also supports a top-down development of a module system by anticipating in the generic import the functionalities of modules yet to be implemented. Generic import is realized by mapping the import interface M_I of the client module to the export interface N_E of a server module which provides the desired resources. In case of several server modules $N_1, \ldots, N_k$ we first have to build their union $N = N_1 \oplus \cdots \oplus N_k$. This is obtained by componentwise union which is disjoint for the body and non-disjoint for export and import.[e]

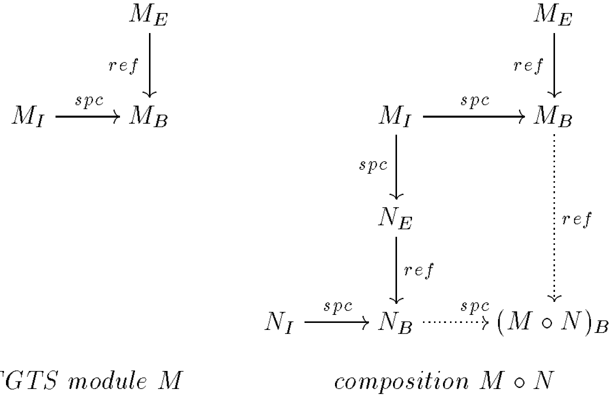


GRACE module M *M imports* $N = N_1 \oplus \cdots \oplus N_k$

The modules of the TGTS concept have almost the same form like in GRACE 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, an operation of module composition is supported. In the diagram below on the right, the composition of modules M and N is shown. The resulting module $M \circ N$ has as import the import of N, as export the export of M, and the body $(M \circ N)_B$ is obtained by gluing the bodies of N and M over the

[e] See [28] for an analysis of different kinds of union of modules with different degrees of sharing in the algebraic module concept [5].

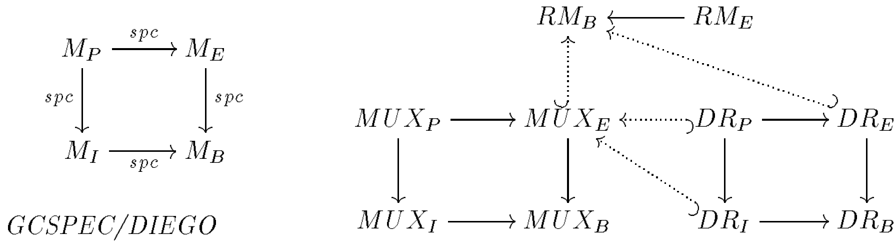
import interface of M. Together with the operation of union discussed above, this allows to build complex module systems while hiding internal import relationships.



TGTS module M *composition $M \circ N$*

In the case of the module system in Section 1.2, this would allow to derive a single module with the export interface of *RM* and the generic import of *MUX* by composing (in this order) *MUX, DR*, and *RM.*

GCSPEC and DIEGO modules (see diagram below on the left) provide, in addition to body, import, and export, a formal parameter interface P that allows, in the presence of a generic import interface, for the distinction between public and hidden import discussed above. Thus, in our notation, P would contain those resources which are generically imported in the export interface.



GCSPEC/DIEGO

The diagram above on the right shows the module architecture in our reference example. Let's try to understand the relation with the textual presentation of modules in Section 1.2 (cf. Figure 1.1 and 1.2). The export RM_E of the top-level module *RM* consists of the types and operations declared and imported under keyword **export**. The body RM_B extends the export by the items of the **body** part. Notice that, in case of a direct import, the imported items

	export	direct import	generic import	hidden import	public import	module semantics
GCSPEC	x		x	x	x	$I \to E$
GRACE	x		x	x		$I \to E$
DIEGO	x		x	x	x	distrib.
PROGRES	x	x		x	x	—
TGTS	x		x	x		E

Table 1.3: Modules: Interfaces, semantics, and interconnection mechanisms.

are formally included in the specification component (i.e., export or body) in order to make it syntactically complete. For the same reason, the body always includes the export.

In the MUX module in the lower left, the parameter MUX_B contains the generic import of the MUX export, i.e., the imported type graph. The import interface MUX_I contains the complete generic import, that is, MUX_P together with the procedures *new, kill, mount*, and *unmount*. The export MUX_E comprises everything which is listed under **export**, and the body is given by the whole module where body rules "overwrite" corresponding interface rules. The DR module is structured in a similar way but for DR_P and DR_I who are not needed for generic import like above but for modeling the *partial import* of DR from MUX: Parameter DR_P consists of the imported MUX types, and export DR_I extends DR_P by the imported procedure *rel*. The relations within the modules MUX and DR are specializations in the sense of the previous section. Export and body of RM are connected by an implementation relation. The arrows of module interconnections, dotted for readability, represent simple inclusions.

The features of the various concepts are summarized in Table 1.3. The first five columns refer to the form of modules as discussed above. The last column lists the (operational) semantics of modules. Here $I \to E$ for GCSPEC and GRACE means that modules define a mapping from their import to their export semantics (i.e., transformation relation). DIEGO modules are given an operational semantics in terms of distributed graph transformation [29] where modules are understood as distributed components. The semantics of TGTS modules is defined as the semantics of their export interface E. No explicit definition is given in [15] for the operational semantics of PROGRES packages. However, like in all concepts where no run-time structure is intended, an operation of flattening can be defined which compiles a system of modules or packages into a flat specification.

1.6 Summary

This chapter provided an informal analysis and classification of module concepts for graph transformation systems as well as a reference example for illustrating the concepts. None of the published approaches supports all the aspects that appear in the example. Hence, we briefly discuss how the reference example could be specified using the features available in the respective (concrete) module concepts.

In order to specify the example in the GRACE module concept, four problems have to be solved. First, parameter passing to procedures is not supported. This can be avoided using an encoding like it is sketched at the end of Section 1.3. Second, the import relation in GRACE does not allow to change the graph types between two different modules. Thus, all modules have to be formulated over the same global type graph, i.e., the union of the type graphs of *MUX* and *DR*. Third, in the import and export interface of GRACE modules only names are allowed. Thus, we can only import and export the names of rules and procedures (instead of interface rules like in *MUX* and *DR*). Finally, GRACE modules do not provide a parameter part, that is, the public imports in *MUX* and *DR* have to be replaced by hidden ones.

The first two problems above apply in a similar way to DIEGO modules. A third problem is the absence of control structures. These have to be encoded into the graphical structure. Due to the limited use of control structures in the example, this seems still feasible here, but in general it can make specifications a lot more complicated.

Also in the PROGRES package concept [15] there is no explicit mechanism for generic import although, according to the UML meta model [30], packages may contain templates, i.e., generic components which may be instantiated by concrete counterparts. In this case, the resources of the generic import of *MUX* may be declared as templates for actual resources. Moreover, since only signatures of procedures may be exported, the rules have to be hidden from the interfaces.

TGTS modules do neither support procedures with parameters nor control structures. The first problem can be solved by encoding as discussed above. The absence of control structures may be compensated by the use of refinements and rule expressions as described in Section 1.4. This option is limited, however, to deterministic control structures, i.e., where the overall (visible) effect of a procedure can be described by a single rule.

In general we noticed a trade-off between more advanced structuring means *in-the-small* and *in-the-large*. E.g., complex procedures make it more difficult to define a satisfactory notion of specialization or refinement. This is one of the

main topics of future research which could result in a module concept integrating the features of different approaches. Interesting results may be expected from the cross fertilization between programming language and software engineering concepts on the one hand and the theory of algebraic specifications on the other hand. Another subject of interest are runtime structuring features like distributed states which are not considered in this paper.
Finally it shall be noted that, since this is not a formal paper, much of the classification here is based on intuition and interpretation on behalf of the authors. Moreover, some of the surveyed concepts are still work in progress and may be subject to change.

Acknowledgment The authors are indebted to Hans-Jörg Kreowski for his detailed comments on a draft version of this paper and to Sabine Kuske, Katharina Mehner, and Francesco Parisi-Presicce for inspiring discussions.

References

1. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
2. H. Ehrig and G. Engels. Towards a module concept for graph transformation systems. Technical Report 93-34, Leiden University (The Netherlands), 1993.
3. H. Ehrig and G. Engels. Pragmatic and semantic aspects of a module concept for graph transformation systems. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 137–154. Springer Verlag, 1996.
4. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1985.
5. 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.
6. H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units - a step into GRACE. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 89 – 106. Springer Verlag, 1996.
7. 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*, 1999. To appear. Also Tech. Rep. 7/96 of University of Bremen.

8. R. Heckel, B. Hoffmann, P. Knirsch, and S. Kuske. Simple modules for GRACE. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformaiton (TAGT'98), Paderborn*, November 1998.
9. 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.
10. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.
11. A. Schürr. Programmed graph replacement systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 479 – 546. World Scientific, 1997.
12. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In this volume.
13. P. Klein, M. Nagl, and A. Schürr. IPSEN tools. In this volume.
14. A.J. Winter and A. Schürr. Modules and updatable graph views for programmed graph rewriting systems. Technical Report 97-3, RWTH Aachen, FG Informatik, October 1997.
15. A. Schürr and A.J. Winter. UML packages for PROgrammed Graph REwrite Systems. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn*, November 1998.
16. G. Booch, J. Rumbaugh, and I. Jacobsen. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
17. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
18. C. Tuijn and M. Gyssens. CGOOD, a categorical graph-oriented object data model. *TCS*, 160:217–239, 1996.
19. M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinements and modules for typed graph transformation systems. In *Proc. Workshop on Algebraic Development Techniques (WADT'98), Lisbon, 1998.* To appear in Springer LNCS, 1999. Also Tech. Rep. SI 98/05, Universitá di Roma *La Sapienza*, Dip. Scienze dell'Informazione.
20. M. Große–Rhode, F. Parisi-Presicce, and M. Simeoni. Spatial and temporal refinement of typed graph transformation systems. In *Proc. Mathematical Foundations of Computer Science (MFCS'98), LNCS 1450*, pages 553–561. Springer Verlag, 1998.

21. M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn*, 1998.
22. R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems.* PhD thesis, TU Berlin, 1998.
23. R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. A view-based approach to system modelling based on open graph transformation systems, 1999. In this volume.
24. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287 – 313, 1996.
25. K.M. Chandy, J. Misra, and L.M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1:144–156, May 1983.
26. A. Schürr. Logic based programmed structure rewriting systems. *Fundamenta Informaticae*, 26(3,4):363 – 386, 1996.
27. H.J. Kreowski, S. Kuske, and A. Schürr. Nested graph transformation units. *Int. Journal of Software and Knowledge Engeneering*, 7(4):470–502, 1997.
28. E.K. Blum and F. Parisi-Presicce. The semantics of shared submodules specifications. In *Theory and Practise of Software (TAPSOFT'85), Vol. 1, LNCS 185*, pages 359–373. Springer Verlag, 1985.
29. G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, TU Berlin, 1996. Shaker Verlag.
30. The Object Management Group. OMG UML Specification, V. 1.3, 1999.

Index

basic specification, 9

composition of modules, 14
control, 9

generic import, 4, 14
graph procedures, 9
graph type, 9

implementation, 10
import, 13

module architecture, 13
module concept, 2

procedural abstraction, 9

refinement, 11
relations of basic specifications, 10

semantics of basic specifications, 9
semantics of modules, 16
specialization, 11

union of modules, 14