

# UML Packages for PROgrammed Graph REwriting Systems

Andy Schürr

Software Engineering Institute, University BW München  
Werner-Heisenberg-Weg 39, D-85577 Neubiberg, Germany  
email: schuerr@informatik.unibw-muenchen.de

Andreas J. Winter

Lehrstuhl für Informatik III, RWTH Aachen  
Ahornstr. 55, D-52074 Aachen, Germany  
email: winter@i3.informatik.rwth-aachen.de

**Abstract:** *Specification and rapid prototyping of graph manipulation software by means of PROgrammed Graph REwriting Systems (PROGRES) is a paradigm, which attracts more and more interest in various fields of computer science. Nowadays produced specifications for process modeling tools, database query languages, etc. have a typical size of about 100 to 300 printed pages. They suffer severely from the lack of any module concept. This paper introduces a module concept for the graph rewriting (transformation) language PROGRES, which is closely related to the package concept of the standardized OO modeling language UML. It supports a variety of software design patterns including the construction of “Abstract Graph Types” and “Updatable Graph Views”.*

**Keywords:** visual programming, graph transformations, module concept, software design styles and patterns

## 1. Introduction

Visual programming languages and tools which are based on the concept of *graph transformations* (GTs) are attracting more and more interest in various fields of applied computer science as well as in related engineering disciplines. They combine two successful principles in one formalism: (1) graphs as a well-understood and popular data model and (2) rules as declarative means for the description of complex transformation and inference processes on complex data structures. Nevertheless, the graph transformation community is still waiting for the breakthrough concerning the applicability of their languages and tools in the real (industrial) world.

The GT language PROGRES [Sch91, Zün96] is for instance used at various sites for specification and rapid prototyping activities. But recently produced specifications of real configuration management, process modeling, reverse engineering, and distributed system analyzing tools usually have a size of more than 100 or even up to 300 printed pages. Keeping these specifications in a consistent state and reusing generic parts of one specification within another specification is a nightmare without the existence of any module concept. Thus the lack of any (implemented) GT module concepts is one of the main hindrances for a wider distribution of GT specification or programming languages in general and the language PROGRES in particular.

This problem should be familiar for software developers of the late 60ies, expert system developers of the late 70ies, and object-oriented (OO) modeling advocates of the 80ies. Well-known software engineering concepts like “abstract data types” [Par72] and “programming in the large” [DK76] have been invented many years ago to overcome these problems. Later on, these concepts have lead to the development of modular programming languages like Ada [Weg80], the development of software design languages like HOOD [Rob72], and to the development of module concepts for knowledge representation languages like PROTOS-L [Bei95].

Rather recently, the OO modeling community made significant progresses concerning the introduction of a module concept, which allows the distributed development of large software analysis and design models and the reuse of once produced submodels. The *Unified Modeling Language* UML [Rat97] developed by Booch, Rumbaugh, and Jacobson is the first OO modeling notation addressing all facets of a state-of-the-art module concept. Its packages build shells around arbitrary types of diagrams or, more general, around arbitrary sets of related declarations.

In the mean time a number of graph transformation researchers joint their efforts in the so-called GRACE initiative [AEH96]. Their common goal is the development of a GRaph Centered Environment, which supports modular programming with various forms of graph data models, rewrite rules, and rule regulation mechanisms. Related publications present our first attempts to introduce import/export as well as inheritance/refinement relationships into the world of GT languages [EE96, HCE96], to encapsulate graph transformation algorithms by means of so-called transformation units [KK96, Sch96], to adapt the concept of database views to graph data types [NS96], and to develop new concepts for the design of distributed systems of graph objects [TS95].

Taking all these sources of inspiration into account, we are currently developing a module concept for PROGRES, which is intentionally kept as simple and flexible as possible. Its draft version, presented here, is closely related to the package concept of UML. More precisely, it is based on the formal definition of a refined UML package concept as introduced in [SW97].

The rest of this paper is organized as follows: very brief introductions to PROGRES and UML’s package concept

are followed by a short presentation of the PROGRES package concept and a survey of supported software design styles (patterns). The paper concludes with a short discussion of related work and the usual summary.

## 2. The Language PROGRES

In this section we give a very brief introduction to the expressive power of the language PROGRES. For a detailed discussion of the language's features and the integrated development environment the reader is referred to [SWZ98, Sch91, Zün96, Nag96].

The language PROGRES is based on the data model of directed attributed graphs and allows to describe transformations on graphs by rules declaratively. It was originally designed for describing internal data structures, operations and relations of tightly integrated Software Engineering tools. Its semantics is formally defined, and it is supported by an integrated set of tools which allow to edit graphically or textually, to analyze, and even to execute specifications. Therefore, PROGRES constitutes an *executable visual language* for specifying systems based on internal graph structures. Together with the ability to generate stand-alone prototypes from specifications the suitability of our Graph Grammar Engineering approach [SWZ95] has been demonstrated for some application areas.

PROGRES specifications consist of two parts: the static *graph schema* and the schema consistent *graph transformations*. The schema allows to express static properties of the regarded class of graph. Its visual notation has the form of EER diagrams with *entities* called node types and *binary relationships* called edge types. Properties of node types, which are not relationships between nodes, are modeled as attributes. A *type hierarchy*, similarly to object-oriented approaches, can be established on these node types. Defining edge types in the schema allows to check statically whether edges are used in a correct context (source and target nodes) in graph transformations. Furthermore cardinalities in the schema and constraints reflect the invariants of the modeled data structure e.g. by restricting the number of source or target nodes that can be reached by traversing edges of certain types.

The upper screen shot in Fig. 3 shows the graph schema of an *Airline Management System*. Its purpose is to provide regular flights between some chosen airports under some given restrictions concerning the available staff and planes. Node types are represented by solid boxes. Dashed arrows depict the *is a* hierarchy and the other edges indicate relationships between nodes of the corresponding types.

Having defined the static graph schema, production rules can now be developed which only allow schema consistent transformations. The rule *ReserveCheapestFlight* in the lower half of Fig. 3 describes a transformation on the underlying graph structure. It makes a reservation for a flight from the given origin to the given

arrival airport provided it has enough available seats for the reservation request and is the cheapest among all flights with these conditions. It consists of a left and a right hand side. The left hand side describes the graph pattern which is matched and replaced by the pattern on the right hand side when the production rule is applied. Note that it is possible to identically replace nodes, denoted by  $x' = x$  which allows to specify the retained context of operations explicitly..

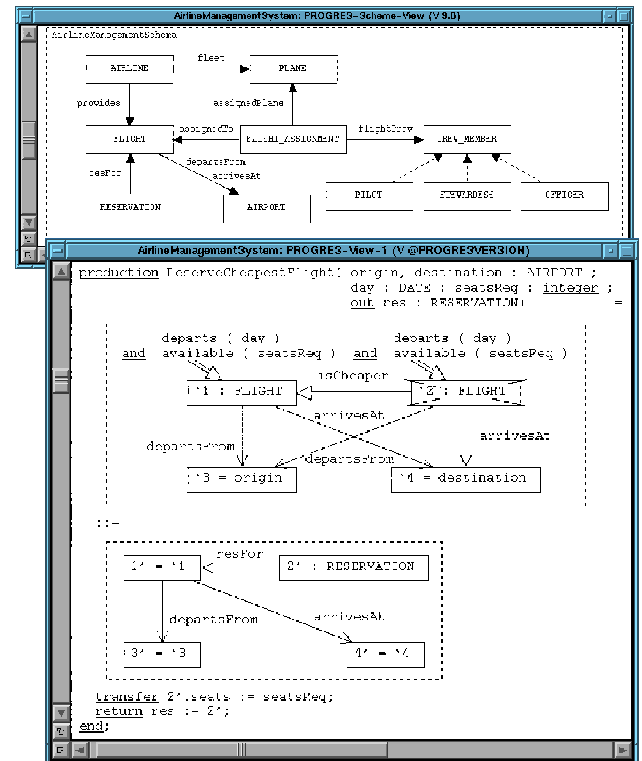


Fig. 1: PROGRES graph schema and production rule

Besides nodes and edges in the production of Fig. 3, the rule's left hand side also refers to forbidden context (crossed-out node) and uses derived graph properties like value restrictions and path navigations (double arrows).

To summarize, PROGRES and its integrated environment define and implement a graph-transformation-based, strongly typed programming language with well-defined syntax, static and dynamic semantics. Being a mixed textual and diagrammatic language, it permits quite different styles of programming, and supports

- graphical as well as textual definition of graph schemata with declaration of derived graph properties,
- rule-oriented and diagrammatic specification of atomic graph transformation steps by means of parametrized graph production rules with complex preconditions, and
- imperative programming of composite graph transformation processes by the means of deterministic and non-deterministic control structures.

### 3. The UML Package Concept

The information hiding and modularization concept of UML packages was strongly influenced by the design of the OO programming languages C++ and Java. It has the following properties:

- A *package* builds a shell around a group of closely related declarations; usually these declarations have the form of a diagram and define the data structures and the operations of a modeled (software) system.
- It is the single purpose of a package to regulate the *visibility* of its declarations, i.e. to restrict the usage of its declarations to well-defined parts of a system model.
- As a consequence, packages have *no run-time semantics* at all, i.e. the semantics of a system model is not changed if all its declarations are put into a single package.
- A *dependency* from a client package to a server package reveals some of the server’s declarations (resources) while others remain hidden.
- *Import* dependencies may be used to access public resources of a server package, whereas *refinement* relationships provide access to its protected resources.
- Furthermore, UML supports *nesting* of packages with inverse scoping rules of block-structured languages, i.e. a parent package has an implicit import dependency to any child package, but not the other way round.
- A package can be developed, compiled, and tested independently from other packages and it may be replaced by another package with the same export interface.

Table 1 explains the interaction between the visibility tag of a package resource — which has a value from the ordered set  $\{ public \geq protected \geq private \}$  — and the two types of package dependencies in more detail.

	public	protected	private
import	visible	invisible	invisible
refine	visible	visible	invisible

**Table 1: Simplified visibility rules of UML**

The UML visibility rules as presented here are simplified due to the fact that they do not take implicit import dependencies as well as visibility tags of package dependencies themselves into account. Please note that UML version 1.1 neither explains how visibility tags of dependencies affect the visibility of imported resources nor their purpose. For further details concerning still existing design flaws of the UML package concept, proposals how to remove these flaws, and a precise formal definition of a complete set of visibility rules the reader is referred to [SW97].

In the following section we will explain our interpretation of the purpose of visibility tags for package dependencies. Furthermore, we will introduce a number of graph-transformation-specific, but not PROGRES-specific package concept extensions.

### 4. The PROGRES Package Concept

Adopting the package concept of UML as the PROGRES modularization concept we have to explain our interpretation of the visibility tags of package dependencies. Later on we have to introduce two additional tags for exported package resources, which allow one to restrict the usage of certain kinds of exported resources in client packages. Furthermore, we have to explain the “semantics” of import and refinement dependencies between packages more precisely as in the UML standard and to fine-tune the standard UML semantics for the purposes of a GT based language.

Combining the two types of package dependencies with three different visibility tags we are able to distinguish six different types of package dependencies:

- *Interface import*: a public import dependency corresponds to the concept of definition module imports in Modula-2. It allows one to import those (node) types from server packages which are used as parameter types of public graph transformation operations of the client package.
- *Protected import*: a protected import dependency has to be used, whenever certain implementation details based on imported resources have to be hidden from regular clients, but must be revealed to refining packages for redefinition purposes.
- *Implementation import*: a private import dependency corresponds to the concept of implementation module imports in Modula-2. It allows one to import all those resources which are used to implement the given package without revealing this fact to the package’s clients.
- *Interface inheritance*: a public refinement dependency defines a kind of subtype relationship between two packages. It is an assertion for all regular clients of these packages that the interface of the refining package is an extension of the interface of the refined package.<sup>1</sup>
- *Protected inheritance*: a protected refinement dependency allows the construction of a kind of subtype relationship between two packages that is invisible to regular clients, but visible for refining packages. It seems to be less useful than the other five presented types of package dependencies.
- *Implementation inheritance*: a private refinement dependency is closely related to the well-known concept of implementation inheritance. It allows one to implement a package as a refinement of another package without any restrictions concerning the export interface of the refining package. It is a matter of debate in the software engineering community whether implementation inheritance is a dangerous or a useful concept.

1. A refining package may redefine inherited operations as long as parameter lists of these operations are not modified and all inherited integrity constraints as well as pre- and postconditions are still observed. These conditions (except for the compatibility of parameter lists) have to be checked at run-time [WS97].

Based on the explanation of six different types of dependencies we are now able to discuss their semantics in more detail. Please remember that packages — as introduced here — do not have a run-time (dynamic) semantics at all, i.e. we are talking about compile-time (static) semantics only. All static semantics rules discussed below restrict the way how exported declarations of a server package may be used in its client packages and how declarations may be decorated with visibility tag values. In particular there are no restrictions at all (except several hundreds of “programming in the small” static semantics rules documented in [Sch91]) concerning the usage of declarations as long as the using declaration is part of the same package.

Before introducing static semantics rules, which restrict the usage of declarations across package boundaries, we have to summarize the available categories of declarations in PROGRES and the ways how these declarations may be used for building new declarations (cf. Sec. 2)<sup>1</sup>:

1. *Operation call*: once defined graph tests or transformations may be used to construct new more complex graph tests or transformations; the new operation calls the already defined graph tests and transformations.
2. *Parameter type*: attribute types and node types may be used as formal parameter types of graph tests and graph transformations.
3. *Schema extension*: attribute types are used for defining attributes of new node types, whereas node types are needed for the declaration of new edge types.
4. *Graph query*: attributes, node types as well as edge types are needed to define the graph patterns of sub-graph tests or application conditions of graph transformations. In both cases these elements are used for inspecting but not for modifying a given (sub-)graph.
5. *Graph modification*: attributes, node types, and edge types are used for constructing the left- and right-hand sides of graph transformations, i.e. for defining necessary graph modifications.
6. *Schema refinement*: new node types may be defined as subtypes of already existing node types. They may redefine inherited attributes of their supertypes.
7. *Operation refinement*: a forthcoming real object-oriented version of PROGRES will not only support the redefinition of attribute evaluation functions, but also the redefinition of graph transformations, which are associated with (complex) node types.

The first four items of the list above allow one to build new *abstract graph types* (packages) on top of already existing

1. Due to lack of space, we do not discuss the following categories of declarations: functions, path expressions, restrictions, and integrity constraints. They are treated in a similar manner as presented declaration categories. Furthermore, the term “transformation” summarizes productions and transactions, whereas the term “query” summarizes tests and queries. Finally, we have dropped the distinction between abstract node classes and concrete node types, as made in PROGRES version 9 [SWZ98].

abstract graph types (packages). A new graph type usually defines its own graph structure and creates connections (edges) between its own nodes and the nodes of the underlying abstract graph type. It manipulates the nodes and edges of the underlying abstract graph type by calling the associated operations only<sup>2</sup>. This leads to a procedural and nonvisual programming style as discussed in [WS97].

In order to preserve the visual flavor of PROGRES specifications it is sometimes useful to permit the graph transformations of one package to create and destroy instances of imported node and edge types directly. This corresponds to item 5 above, where we allow the unrestricted usage of (imported) node and edge types in the left- and right-hand side of graph transformation rules (productions). It is an important decision for the design of a package whether its export interface supports direct manipulation of exported node and edge types. Its implementation must be prepared for keeping the exported and externally modifiable parts of its graph structure consistent with its hidden parts. The reader is referred to [WS97] for a detailed discussion of how this kind of *view update problem* may be solved using active integrity constraints.

The discussion above shows that we need additional means to distinguish between the export of node types, edge types, and attributes for the definition of parameters, schema extensions, and graph queries on one hand, and the export of these resources for direct graph modification purposes on the other hand. As a consequence we are introducing an additional tag for exported resources (declarations) with two possible values:

*mutable* = *m* (the default) and *immutable* = *i*

Mutable resources may be used without any restrictions, whereas imported immutable resources may not be used for direct graph manipulation purposes in client packages (the tag does not impose any restrictions concerning the usage of a declaration inside its own package).

Similarly it is not always useful to allow the redefinition of exported attributes and graph transformations as well as the definition of new subtypes of an exported node type. As a consequence we have to introduce another tag for exported resources, which has the two possible values:

*redefinable* = *r* (the default) and *final* = *f*

Redefinable resources may be used without any restrictions, whereas imported final resources may not be used for redefinition purposes in client packages (the tag does not impose any restrictions concerning the usage of a declaration inside its own package).

2. Graph transformations of the new graph type have the permission to use node and edge types of the underlying graph type for the definition of context conditions and embedding rules, but they may not delete or create instances of these types. Otherwise, it would not be possible to create connections between the nodes of the new graph type and the nodes of the underlying graph type.

Table 2 summarizes the interaction between the visible, the modifiable, and the redefinable tag of exported package declarations and the two main categories of package dependencies. It was constructed as follows: Table 1 was used to determine whether a resource of one package is visible for its dependent packages. Private resources of a package are for instance never visible for the outside world. This has the consequence that all entries of the *private* column of Table 1 are blank. All cells of Table 1 with the value *visible* had to be split in four subcases in accordance with the four possible combinations of the previously introduced modifiable and redefinable tags.

export tags	public				protected				private
	mr	mf	ir	if	mr	mf	ir	if	—
import	mf	mf	if	if	—	—	—	—	—
refine	mr	mf	mr	mf	mr	mf	ir	if	—

**Table 2: Visibility rules of PROGRES**

(m = modifiable, i = immutable, r = redefinable, f = final)

A *public* export which has e.g. a tag  $m = \text{modifiable}$  and a tag  $r = \text{redefinable}$  may be used by its client packages with import dependencies for  $m = \text{modification}$ , i.e. direct manipulation purposes. Furthermore its  $r = \text{redefinable}$  tag is downgraded to  $f = \text{final}$  due to the fact that regular client packages never have the permission to refine the imported graph schema or the imported operations. All remaining cells of Table 2 have to be interpreted accordingly.

The definition of the entries of Table 2 was rather straightforward, with one exception: The *immutable* tag of *redefinable* as well as *final public* exports is only valid for regular import dependencies. It is upgraded to the value *mutable* in the case of a refinement dependency. Otherwise it would be necessary to introduce a fourth export tag which allows one distinguish between

*immutable* for import and refinement dependency

and

*immutable* for import but *mutable* for refinement.

Due to the fact that refining packages usually have to manipulate instances of inherited node and edge types directly and the fact that *immutable*, *final* resources may be moved to a separate package, we made the decision that the *immutable* flag of public resources is ignored by refinement dependencies between packages. It is a matter of debate whether the *immutable* flag of *protected* export resources should be ignored, too. In this case the entries for columns *protected ir* and *if* and row *refine* should be changed from the presented values

$ir = \text{immutable redefinable}$  and  $if = \text{immutable final}$

to the new values

$mr = \text{mutable redefinable}$  and  $mf = \text{mutable final}$ .

Such a modification of the presented visibility rules will be taken into account if and only if external users of the package concept implementation start to complain about the currently made design decisions.

Having explained the interaction between the three different tags of exported package resources and the two different types of package dependencies it is now time to discuss the restrictions concerning the selection of certain tag values. Due to lack of space it is not possible to provide the reader with a complete list of all related static semantics rules, i.e. the following list items presents the most important rules only:

- The visibility tag value of an export operation may not be higher than the visibility tag value of its parameter types. A public operation with private parameter type is one example of a forbidden combination of values.
- The visibility tag value of an edge type or attribute declaration may not be higher than the visibility tag value of the referenced node types. It is for instance useless to define a public attribute for a protected node type or a protected edge type between private node types.
- The supertypes of a node type may not have a lower visibility value than the regarded node type itself. Otherwise, it would be possible to export type hierarchies with “holes”, i.e. to hide subtype relationships between externally visible node types.
- Mutable supertypes may not have immutable subtypes. Otherwise, it would be possible to destroy (create) instances of immutable node types by handling them as indirect members of the mutable supertype.

The presented static semantics rules are not able to prohibit *unresolvable inheritance conflicts*: Let us assume that a package  $P$  declares a public node type  $A$  with a private attribute  $a := e$  as well as two public subtypes  $A1$  and  $A2$ , which redefine the initial value of attribute  $a$  from  $e$  to  $e1$  and  $e2$ , respectively. Furthermore, assume that another package  $Q$  imports the node types  $A1$  and  $A2$  for constructing a type  $B$  is  $_a A1, A2$ . The new node type  $B$  inherits the attribute  $a$  with initial value  $e1$  from  $A1$  and with the initial value  $e2$  from  $A2$ . The only way to resolve the inheritance conflict, the introduction of a new initial value for attribute  $a$  in subtype  $B$ , is blocked due to the fact that the private attribute  $a$  of package  $P$  is invisible for package  $Q$ .

Currently, we have no good idea how to avoid this problem without introducing very restrictive rules such as

*public or protected node types may not have private redefinable attributes*

or

*the concept of multiple inheritance may not be used across package boundaries.*

Please note that Java avoids this problem by disallowing multiple inheritance of implementations, whereas C++ simply ignore the discussed problem due to absence of a properly defined package concept.

## 5. Specification in the Large Patterns

Having presented the most important “screws” of our package concept and their meaning it is now time to show how they may and should be used in practice. It is the purpose of this section to demonstrate that the introduced concepts allow one to apply a number of rather different software design styles for structuring large specifications. Due to lack of space we are not able to present detailed examples, but we will explain some important design patterns on a rather abstract level. Please note that it may sometimes be necessary to use combinations of the presented patterns for establishing more complex dependencies between client and server packages.

First of all we have to prove that PROGRES packages are the appropriate means for constructing abstract (graph) data types in the usual sense [Par72]. Fig. 2 sketches the relationships between a server package that realizes an abstract graph type and a client package that constructs a new (abstract) graph type using the imported graph type.

The server package exports a number of public, immutable, and final declarations. Some declarations define the constructed graph type’s schema, others the accompanying graph transformation operations. The schema exported by the server package may consist of a number of node type and edge type declarations. The classical abstract graph type consists of a single *immutable* node type declaration, though. This type declaration as well as the associated interface operations have a *final* tag if the construction of subtypes should be prevented. These subtypes would have the permission to redefine and thereby to alter the exported operations and the underlying graph structure.

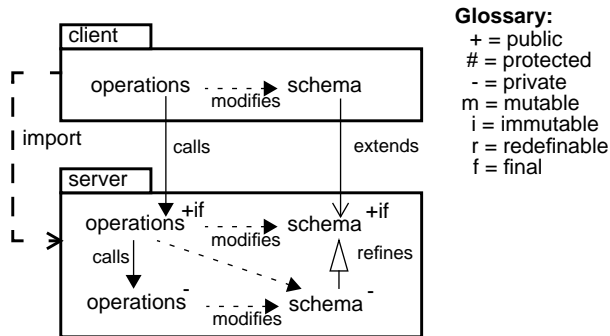


Fig. 2: Full data abstraction pattern

The revelation of the hidden complex graph structure which is necessary for implementing the functional behavior of the package is done in the private part of the abstract graph type. It extends and refines the public part of the schema. That means, the concept of aggregation is simulated by choosing one node type as the only visible representative for the underlying realization. The servers’ exported operations invoke the hidden operations in the private part of the package. They perform the transformation on the internal graph structure. That means, access to

the internal graph structure is only possible by calling the appropriate exported interface operations of the abstract graph type. In this way the package can ensure that the internal graph structure remains in a consistent state because a client does not have any possibility to circumvent the prepared interface operations which perform well-defined transformations only. Consequently, the operations in the client package essentially consist of calls of interface operations exported by other packages.

The following Fig. 3 explains how it is possible to define a “semi-abstract” graph type. It exports a larger part of its graph schema. Marking the public schema also with the *mutable* flag allows its clients to manipulate instances of visible node and edge type declarations directly. The depicted server package constructs a kind of updatable graph view without any exported operations. The client package uses its own graph transformations to manipulate the visible part of the graph structure of the server package. In contrast to the encapsulated abstract graph type in Fig. 2 the client is not restricted to call exported operations of the server package only. Instead the client is able to define its graph transformations in a visual way by having access to instances of the server’s schema. These graph transformations are even allowed to create and destroy instances of the imported schema.

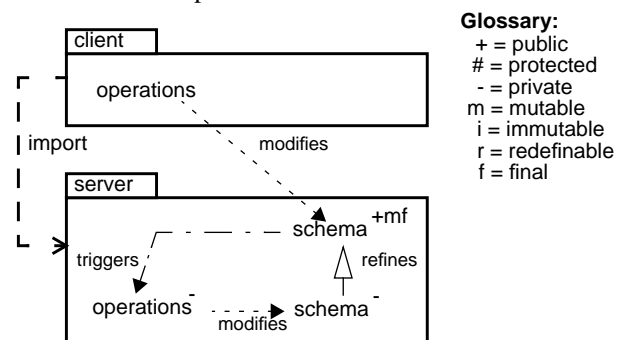


Fig. 3: Updatable view definition pattern

These graph manipulations of the client may introduce inconsistencies between the visible and the hidden part of the server’s graph structure. Inconsistencies may occur because the client does not see the internal graph structures of the server package. Therefore, it is not possible to ensure the preservation of the consistent context for direct graph manipulations performed by the client. For example, the server could provide an internal graph structure which is responsible for locating certain elements in the graph efficiently which is not visible to the outside world. The client is not able to access the internal structure and, consequently, e.g. created elements can not be integrated with the efficient access data structure.

Active integrity constraints, which are very similar to ECA-rules of active database systems [WC96], are responsible for detecting introduced inconsistencies and for taking appropriate repair actions in the server package. If a

constraint detects a certain graph pattern which corresponds to an inconsistent execution state a hidden operation is triggered. This operation is responsible for reestablishing consistency between the visible and externally modifiable part of the implemented graph structure and the associated hidden parts [WS97] if possible.

The last pattern presented in Fig. 4 sketches the construction of subtypes of abstract graph types. It is more or less a new variant of the pattern for abstract graph types in Fig. 2. In addition the client is allowed to refine the schema and the inherited operations. Please note that it is a matter of debate whether the operations of the client package should be allowed to manipulate instances of the inherited type definitions of the server package as depicted in Fig. 4. The main problem is that the client package is always able to circumvent the restriction to manipulate instances of inherited node types directly by deriving a new locally modifiable node type from the inherited unmodifiable node type. This is one of the reasons to upgrade the *immutable* flag to *mutable* for inherited public resources also marked with the *redefinable* flag (cf. Table 2 of Sec. 4).

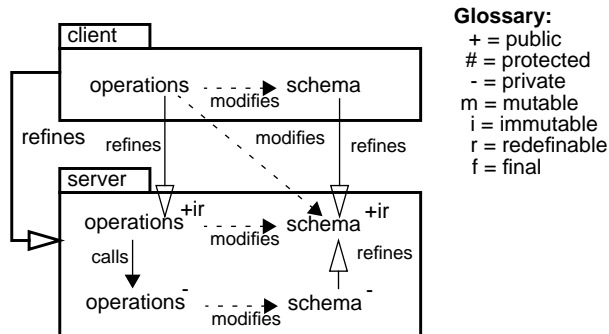


Fig. 4: Subtype definition pattern

The design pattern of Fig. 2 with a *public* import dependency to a refinable server package is for instance closely related to the introduction of a *parameter part* in [EE96], which is shared by the import and the export part of a given package. And the design pattern of Fig. 4 with a *private* refinement dependency corresponds to what is usually called *implementation inheritance*. Taking into account

- that it is also possible to nest packages in order build complex subsystems with their own subarchitectures
- and that a client package is usually implemented on top of more than one server package
- and that a server package is often used by different client packages in different ways

it should be rather obvious that the PROGRES package concept supports an overwhelming variety of different software design styles and patterns. It is the subject of future research activities to distinguish good graph transformation system design patterns from bad ones and to invent new patterns for more specific purposes. It is still an open question to which extent all those patterns presented in object-oriented design pattern books such as [GHJ95] or

[Fow97] are also appropriate for the construction of large graph transformation systems.

## 6. Summary

The language PROGRES and its tools are the results of many years of application-oriented graph grammar or graph transformation research activities. Nowadays they are used at various sites — as e.g. at Carleton University in Ottawa, INRIA in Sophia Antopolis, or Gesamthochschule Paderborn — for specifying and prototyping software, database, and knowledge engineering tools.

Despite our success in demonstrating the usefulness of graph grammar engineering concepts and tools for software development, the currently available PROGRES release<sup>1</sup> is not yet ripe for real industrial software development projects. Compared with the history of imperative programming languages, PROGRES has reached the state of languages like Algol-68 or Pascal. It has a sophisticated type system and enforces a well-structured style of programming, but gives not yet any support for “specification in the large” activities.

Considering the pile of already published module concept papers in the last 30 years, it was and is clearly not our goal to start from scratch and to reinvent well-known “programming in the large” concepts. On the contrary, we started the development of the PROGRES module concept based on our experiences with the design of a module interconnection language MIL [Nag90] and recently made efforts in the object-oriented world to introduce a package concept for the Unified Modeling Language UML [Rat97].

Modules in the sense of MIL and packages in the sense of UML are establishing visibility boundaries around arbitrary sets of related type and operation declarations. They give their constructors the flexibility to adhere to quite different popular *software design styles*:

- Packages that export a single (not directly modifiable) “main” node type and a number of related operations define abstract data types in the classical sense [Par72].
- Packages with type declarations only in their interfaces and without any associated operations are our means for decomposing large graph schemata in the same way as in the database management system PCTE [Ecm90].
- Packages that export a number of directly modifiable types together with carefully defined active constraints are useful for the construction of updatable graph views, which may not only be manipulated using a number of predefined graph transformations as suggested in [EHT97].
- Packages that export a single transformation at their interfaces together with an appropriate set of graph schema declarations allow the definition of graph transformation units as suggested in [KK96].

1. For further details concerning PROGRES release version 9 cf.: <http://www-i3.informatik.rwth-aachen.de/research/progres>

- Packages that summarize and propagate the exported resources of a number of imported packages to their own interfaces offer more or less the concept of horizontally structured graph types as presented in [HCE96].
- Packages which refine the graph schemata and operations of a number of related packages offer more or less the concept of vertically structured graph types as presented in [HCE96].

Having completed the design of the PROGRES package concept as presented here, it is our plan to finish the implementation and to provide a graph transformation language release that supports “specification in the large” activities until the end of this year.

Long lasting efforts are still necessary to develop all the details of a more ambitious package concept that supports the construction of hierarchical and distributed graph transformation systems as well as automatic gluing of independently constructed redefinitions of graph transformation rules. Finally, we have to emphasize that we are just starting to assemble more knowledge about useful graph transformation system design patterns and their relationships to pattern catalogues as presented in object-oriented design books.

## References

- [AEH96] M. Andries, G. Engels, A. Habel, B. Hoffmann, H. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer: *Graph Transformation for Specification and Programming*. Technical Report 7, University of Bremen, 1996.
- [Bei95] C. Beierle: *Concepts, Implementation, and Applications of a Typed Logic Programming Language*. In Beierle and Plümer, editors: *Logic Programming: Formal Methods and Practical Applications*. Vol. 11 of *Studies in Computer Science and Artificial Intelligence*, chapter 5, Elsevier Science B.V., 1995.
- [CEE96] J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors: *Proc. 5th Int. Workshop on Graph Grammars and Their Application to Computer Science*: LNCS 1073, Springer, Berlin, 1996.
- [DK76] F. DeRemer and H. Kron: *Programming--in--the--large versus Programming--in--the small*. *IEEE Transactions on Software Engineering*, SE-2(2):80--86, June 1976.
- [Ecm90] European Computer Manufacturers Association (ECMA): *ECMA Standard 149: Portable Common Tool Environment (PCTE) Abstract Specification*, 1990.
- [EE96] H. Ehrig and G. Engels: *Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems*. In Cuny et al. [CEE96], pages 137--154.
- [EHT97] G. Engels, R. Heckel, G. Taentzer, H. Ehrig: *A View-Oriented Approach to System Modelling Using Graph Transformations*. In M. Jazayeri and H. Schauer, editors: *Proceedings European Software Engineering Conference (ESEC) '97*. LNCS 1301, pages 327-343, Springer, Berlin, 1997.
- [Fow97] M. Fowler: *Analysis Patterns*. Addison Wesley, Reading, MA, 1997.
- [GHJ95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [HCE96] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe: *Horizontal and Vertical Structuring in Typed Graph Transformation Systems*. *Mathematic Structures in Computer Science*, 6(6):613--648, December 1996.
- [KK96] H.-J. Kreowski and S. Kuske: *On the Interleaving Semantics of Transformation Units --- A Step into GRACE*. In Cuny et al. [CEE96], pages 89--106.
- [Nag90] M. Nagl: *Software Engineering: Methodological Programming in the Large*. Springer, Berlin, 1990 (in German).
- [Nag96] M. Nagl, editor: *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. LNCS 1170. Springer, Berlin, 1996.
- [NS96] M. Nagl and A. Schürr: *Software Integration Problems and Coupling of Graph Grammar Specifications*. In Cuny et al. [CEE96], pages 155--169.
- [Par72] D. Parnas: *A Technique for Software Module Specifications with Examples*. In *Comm. of the ACM*, volume 15, pages 330--336. ACM Press, 1972.
- [Rat97] RATIONAL SOFTWARE CORPORATION: *UML Semantics*. <http://www.rational.com/uml>.
- [Rob92] P. Robinson: *Hierarchical Object-Oriented Design*. Prentice Hall, Engelwood Cliffs, NJ, 1992.
- [Sch91] A. Schürr: *Operational Specification with Programmed Graph REwriting Systems*. Deutscher Universitäts-Verlag, Wiesbaden, 1991 (in German).
- [Sch96] A. Schürr: *Programmed Graph Transformations and Graph Transformation Units in GRACE*. In Cuny et al. [CEE96], pages 122--136.
- [SW97] A. Schürr and A. Winter: *Formal Definition and Refinement of UML's Module/Package Concept*. In J. Bosch and S. Mitchell, editors: *Object-Oriented Technology - ECOOP '97 Workshop Reader*. LNCS 1357, pages 211-215, Springer, Berlin, 1997.
- [SWZ95] A. Schürr, A. Winter, and A. Zündorf: *Graph Grammar Engineering with PROGRES*. In W. Schäfer and P. Botella, editors: *Proceedings European Software Engineering Conference (ESEC) '95*. LNCS 989, pages 219-234, Springer, Berlin, 1995.
- [SWZ98] A. Schürr, A. Winter, and A. Zündorf: *PROGRES*. In G. Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Specification and Programming*. World Scientific, Singapore, 1997.
- [TS95] G. Taentzer and A. Schürr: *DIEGO, another step towards a module concept for graph transformation systems*. In A. Corradini and U. Montanari, editors: *Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRA'95)*. Vol. 2 of *ENTSC*. Elsevier Science B.V., 1995.
- [Weg80] P. Wegner: *Programming with Ada*. Prentice Hall, Engelwood Cliffs, NJ, 1980.
- [WC96] J. Widom and S. Ceri, editors: *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, CA, 1996.
- [WS97] A. Winter and A. Schürr: *Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems*. Technical Report 97-3, RWTH Aachen, 1997.
- [Zün96] A. Zündorf: *A Development Environment for Programmed Graph REwriting Systems*. Deutscher Universitäts-Verlag, Wiesbaden, 1996 (in German).