Formal Definition of UML's Package Concept

A. Schürr, A. Winter[1]

Lehrstuhl für Informatik III,
RWTH Aachen, Ahornstr. 55, D-52074 Aachen

Abstract: UML is the first OO modeling language with a useful modularization and information hiding concept. It supports nesting, import, and refinement of so-called packages. This paper translates UML's informal package definition into predicate logic formulas and solves some open problems concerning the visibility of exported and imported modeling elements.

Furthermore, the formulas will be transformed into a specification based on PROgrammed Graph REwriting Systems. We will very briefly sketch how a graphical UML editor can be generated from this PROGRES specification.

1 Introduction

After about 20 years of research and development, object-oriented (OO) modeling methods and notations have reached a certain degree of maturity and acceptance. They are no longer the occult science of a small number of OO gurus, but the widely accepted approach for analysis (OOA) and design (OOD) of software (hardware) systems. Popular OO methods — like Booch (1994), OMT (Rumbaugh et al. (1991)), or OOSE (Jacobson (1994)) — are nowadays used to develop systems of continuously increasing size and complexity. As a consequence, produced analysis or design documents may consist of hundreds or even thousands of elements (classes, relationships, etc.). Keeping these documents in a consistent state or reusing generic parts of one analysis or design document within another one is a nightmare without the existence of any module concept.

These problems should be familiar for software developers of the late 60ies. Well-known software engineering concepts like "abstract data types" (Parnas (1972)) and "programming-in-the-large" (DeRemer and Kron (1976)) have been invented to overcome these problems. They lead to the development of modular programming languages like Modula-2 or Ada (Wiener and Sincovec (1984)) and software design languages like HOOD (Robinson (1992)) or EMIL (Börstler (1994)).

For a long time these ideas did not have any significant impact onto the development of OOA/OOD notations. The first generation of OO methods simply ignored the problem. Later on developed approaches offered more or less ad hoc solutions for partitioning analysis and design documents (diagrams) into surveyable pieces. OMT (Rumbaugh et al. (1991)) offers for instance so-called "modules", which allow to decompose unmanageable

[1]This work has been supported by APPLIGRAPH.

diagrams into a number of related diagrams. But there are no means to construct exports or imports of modules. As a consequence, any two elements in two different diagrams with the same name have to be identified. And even more elaborate concepts like categories in Booch (1994), collaborating subsystems with their contracts in Wirfs-Brock et al. (1990), or subsystems with well-defined interfaces in ADM3 (Firesmith (1993)) do not study the interactions between information hiding, module boundary crossing associations and inheritance.

The proposal of a Unified Modeling Language (UML) presented in Rational (1997) as a successor of Booch, OMT, and OOSE, is in our opinion the first OO notation which addresses all facets of a state-of-the-art module concept. Its modules, called packages, build shells around arbitrary types of diagrams (static structure diagrams, collaboration diagrams etc.). They provide a grouping mechanism for their elements. Within packages, an element can be used as source or target in relationships if it is either owned by the package or if the package references the element from another package. In order to be able to reference elements from another package, the client package must have a corresponding dependency to the package which makes the target element accessible. This dependency on package level can either be an import or a generalization dependency.

In the following section we describe the concept of packages in more detail and we show some examples for ambiguous and even contradictive statements concerning the semantics of packages in the UML description. Then, we translate the natural language definition of UML's module concept into predicate logic formulas which give precise answers to the problems raised before.

In Schürr and Winter (1997a) we have already presented formulas for UML 1.0 very briefly. In this paper we deal with UML 1.1 and motivate our reservation with respect to UML semantics with some examples. Finally, we transform the formulas into a formal specification based on graph rewriting systems. The PROGRES specification language, environment and prototyping machinery (cf. Schürr et al. (1995)) allows to generate a graphical UML (class diagram) editor from the specification which "recognizes" inconsistent diagrams to some degree.

2 Informal Description and Problems

The UML package concept offers an information hiding concept which was heavily influenced by the design of C++ (cf. Ellis and Stroustrup (1994)). There may be relationships between elements contained in the same package, but not a priori between an element in one package and an element outside the package. All elements inside a package are featured with a visibility value from the ordered set {public > protected > private}[1].

[1] Please note that both, the `friend` relationship and the `implementation` visibility introduced in version 1.0 of UML, do not belong to the core concepts in UML 1.1 anymore.

There are two ways of making elements outside a package available inside the package:

- Explicit *import* relationships have to be used to access public elements of one package within another package,
- *Refinement* (generalization) relationships provide additional access to protected elements.

Furthermore, UML allows nesting of packages. This concept is rather similar to nested scopes in programming languages à la Modula-2 or Ada.

All concepts are essentially defined in natural language. Although the semantic description for UML 1.1 is accompanied by the UML meta model and Object Constraint Language expressions, there are still many statements which lack the precise definition. We will now sketch our main problems with UML's module concept together with some examples.

In some cases the introduced terms are described in an ambiguous way. This makes it sometimes really difficult to determine their precise semantics. Consider, for instance, a sentence like

> *"Elements made available to another package by the use of generalization ... have the same visibility in the heir as they have in the owning package."*

on page 135[2]. This statement appears to be rather clear on the first sight. But even small examples raise questions which are hardly answered by sentences like this.

a) simple scenario

b) ambiguous scenario

Figure 1: Problem with ambiguous UML semantics.

On the left side of figure 1 we have depicted a situation, where a package `R` uses the element `A`, e.g. a class, made available by generalization from

[2]Page numbers refer to Rational (1997).

package Q. In UML Q is called the *owner* of A, whereas R *references* A. Local references of elements are used to avoid package crossing dependencies to elements located in different packages whenever an element is used in a new context, e.g. to establish a new association to a class defined somewhere else. References carry the path (of packages) to the their source which is Q::A for the reference A in R. All elements must be owned by exactly one package[3]. The package O serves this purpose for Q and R. Note that every element is featured with a visibility attribute. In diagrams the feasible visibility values are represented by single characters ('+' for public, '#' for protected, and '-' for private visibility). If not present, visibilities default to *public*; this is also true for dependencies.

Taking the scenario in figure 1a alone, the sentence above states that the visibility of reference A in package R should be the same as in Q which is protected.

In 1b) the situation is slightly more complex. Here, Q is not the owner of the element A. It is imported from another package P. Package P owns the element A with public visibility. The reference in Q has restricted the visibility of A to protected.

Read the sentence above once more with the new situation in mind. Now, it is not clear which visibility is feasible for the reference A in R (expressed by the question mark). In the example on the left side we were quite convinced that it makes sense to give the visibility in the heir the same value as in the target of the generalization dependency. But, in this case, we have to take into account that the target of the generalization dependency and the owner of the regarded element are not the same package. Taking the sentence above literally, we would have to decide that the visibility of A in R has to be the same as in its owning package P, which is public.

In order to solve this problem, it would be necessary to distinguish that a package does not only make available its own elements by generalization, but also imported elements from other packages.

Until now, we have not even considered the fact that dependencies between packages are also elements and, therefore, carry a visibility attribute themselves. For showing the ambiguity in the UML semantics we did not exploit this feature, yet.

In figure 2 we almost have the same situation compared to figure 1a. According to the previous discussion, we would expect the visibility of the reference A in package R to inherit the visibility property from package Q. In this case Q is both, provider of its elements for heir R and, at the same time, owner of these elements. Therefore, we do not have the problem mentioned above.

The difference of the two figures 1a and 2 is the visibility attribute of the generalization dependency itself. Import and generalization relationships between packages are indirect subclasses of the UML top-level class Element and inherit its property to possess a visibility attribute. We have changed its value from the default public to private. The purpose is to hide the

[3]Obviously the toplevel package in the model is an exception from this rule.

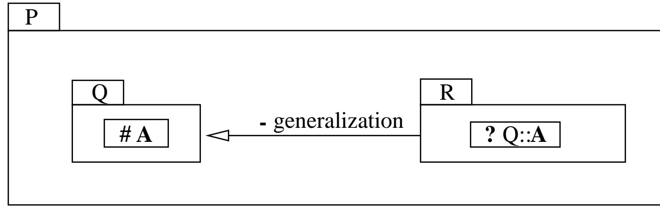

Figure 2: Contradicting UML semantics.

generalization for the outside world. We did not find a single line that explains the consequences of dependency visibilities for connected packages, although the usage of import and generalization relationships with varying visibility values makes sense from the software engineering point of view.

Concerning visibility, a look at page 142 of the UML semantics description revealed the following statement about the issue of private generalization giving us a new source of confusion.

> *"Private inheritance (generalization) ... hides the inherited features of a class (package) ...."*

Although this statement is made for classes, statements concerning generalization should hold for both, classes and packages. Obviously this sentence forces elements which are made available by a private generalization to be visible at the same level as the generalization itself. This means that, whenever the dependency is private and, therefore, not at all visible to the outside world, the elements made available by the generalization should also be hidden. Nevertheless, both citations from the UML semantics report contradict each other in an obvious way.

We have found even more contradictions in the semantics description of UML and we are sure that some more lurk in its over 150 pages. Some of them are obvious remainders of former versions and some of them are not really critical. But, anyway, the aim of a semantics description document should be to make things more precise, not to spread confusion among its readers.

Moreover, we were not able to find some strictly necessary constraints like *"a client A of another (imported/referenced) package B should not add (own) an import relationship from B to another package C"*. They are either too obvious to be part of the UML semantics definition itself or carefully hidden in one of its paragraphs.

Consequently, we introduce predicate logic formulas explained in the following which are our attempt to give precise answers to problems outlined above. Furthermore, we suggest five additional formulas that address still missing restrictions.

Within all these formulas we use the following (all-quantified) variables which correspond directly to the UML semantics description:

$$\begin{aligned} P, P', P'' &\in \texttt{Package is_a Element}, E \in \texttt{Element} \\ \texttt{dep} &\in \{\texttt{imports}, \texttt{generalizes}\} \\ \omega, \omega', \omega'' &\in \{\texttt{public} > \texttt{protected} > \texttt{private}\} \end{aligned}$$

3 Aggregation of Elements and Packages

Any UML document contains a number of top-level packages which represent the regarded system model. Each package defines a visibility shell around a number of elements, which are either

(a) basic constructs of a certain type of diagrams or

(b) nested packages or

(c) dependencies between nested packages.

A package `A` contains an element `E`, if it `owns` or `references` (uses) `E`, which belongs to another package `B`. In the latter case, where `A` references `E`, `E` must be visible inside `A`. Depending on the relationship of `A` and `B` on package level, `A` can "see" elements in `B`. An import relationship from `A` to `B` reveals the public elements in `B`. A generalization makes available the protected elements from `B` in `A` in addition to the public elements. This situation is captured by the following five predicate logic formulas. The interactions between visibility of elements and (explicit) import relationships as well as generalization are the subject of sections 4 and 5, respectively.
We do hope that almost all of them are self-explanatory as soon as the role of ω variable is clear. Terms like

$$P \text{ owns } \omega\ E \text{ or } P \text{ references } \omega\ E \text{ or } P \text{ contains } \omega\ E \text{ or } \ldots$$

have to be interpreted as "Package `P` owns/references/contains/... the element `E` with the visibility $\omega \in \{\texttt{public} > \texttt{protected} > \texttt{private}\}$".

(1) The owner of any element is unique:

$$P \text{ owns } \omega\ E \ \wedge\ P' \text{ owns } \omega'\ E \Rightarrow P = P' \ \wedge\ \omega = \omega'$$

(2) Elements (from other packages) may be referenced if visible. The visibility of the reference may be restricted:

$$P \text{ references } \omega\ E \Rightarrow P \text{ sees } \omega'\ E \ \wedge\ \omega < \omega'$$

(3) The contains relationship comprises owns and reference relationship:

$$P \text{ contains } \omega\ E :\Leftrightarrow P \text{ owns } \omega\ E \ \vee\ P \text{ references } \omega\ E$$

(4) The offers relationship is the transitive closure of contains relationship:

$$P \text{ offers } \omega\ E \quad :\Leftrightarrow \quad P \text{ contains } \omega\ E\ \vee$$
$$\exists P' : P \text{ contains } \omega\ P'\ \wedge\ P' \text{ offers public } E$$

(5a) Visibility of elements (of nested packages) is determined as follows:

$$P \text{ sees } \omega\ E :\Leftrightarrow P \text{ offers } \omega\ E \ldots$$

The first two formulas establish requirements on the concepts of *owns*, *references* concerning the correctness of UML diagrams. Furthermore, the new terms *contains*, *offers*, and *sees* are introduced in order to keep the formulas as simple as possible. Please note that formula (2) uses the definition of (5a) which only considers visibility of nested package elements. We will extend the rather complex definition of the `sees` relationship step by step. Note that these extension have again impact on the validity of references in formula (2). This makes it necessary to take all formulas into consideration.

The most important consequence of the definitions above is that a package sees all public elements of contained packages with the visibility of the containership itself. This is also true for packages which are transitively nested. Elements which are contained within several layers of nested packages can also be offered to the outside world as long as the element is made available to the top-level package as public element.

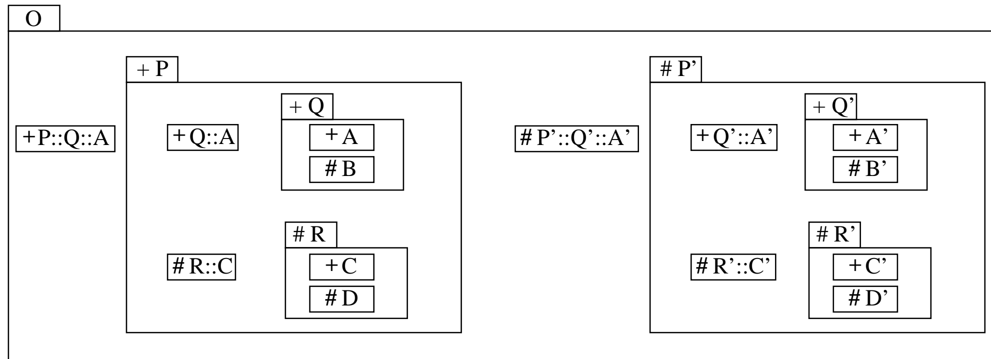

Figure 3: Visibility in nested package.

In figure 3 package `O` contains two packages `P` and `P'`. `P` has public visibility, the visibility of `P'` is protected. `P` (`P'`) contains two packages `Q` and `R` (`Q'` and `R'`). The visibility of `Q` and `Q'` (`R` and `R'`) is public (protected). Each of the packages `Q`, `Q'`, `R`, and `R'` contain two elements, one of which is public, the other one is protected. In this example we can observe how elements of different visibilities propagate upward through nesting levels. The main rule is that only public elements are visible one package level higher. The (new) visibility of the referenced element can be restricted by the package

itself. Here, e.g. the protected element `B` in package `Q` is not visible for `P`. But element `C` is of course visible for `P`, even though the reference `R::C` is restricted by the visibility of `R`, which is protected. Eventually, from all elements `A-E` and `A'-E'` only `A` and `A'` are visible in `O`. Note the different visibilities of the elements which have impact, either for the owner of `O` or an importing client, which will only see public elements in `O`.

Technically, the requirement of `Q::A` being public in `P`, in order to be visible from `O`, is due to the fact that nesting establishes an *implicit import* of the nested packages. We will consider the explicit import relationship between packages in detail in the next section.

4 Export/Import of Packages

We have seen that nesting of packages gives surrounding packages access to the public elements of enclosed packages. Therefore, UML states that nesting establishes a kind of implicit import. Using implicit imports only, a package would never be able to reference elements at the (public) interface of sibling packages. Therefore, UML introduced the concept of explicit import as a dependency relationship between packages that belongs to the common surrounding package of the related client (source) and server (target) package.

These import dependencies even have their own visibility attributes. A *public import*, on one hand, represents a kind of interface import, which is visible for all clients of the surrounding package. A *private import*, on the other hand, is an always hidden import, representing local analysis or design decisions.

The following four formulas are our attempt to formalize imports and exports of packages. Please note that packages do not have any means to define sets of *exported elements*, explicitly. Their public/protected exports are always implicitly determined by their sets of public/protected visible elements. As a consequence, packages do not only export own elements, but also referenced elements from other packages. This takes from client packages the cumbersome burden to import all those elements of other packages which are used in the interfaces of already imported packages.

(6a) Implicit import exists to nested packages and to transitively imported packages (will be extended in the following section):

$$P \text{ imp_imports } \omega\ P' :\Leftrightarrow\ P \text{ contains } \omega\ P' \vee$$
$$\exists P'' :\ (P \text{ contains } \omega\, P'' \vee P \text{ exp_imports } \omega\, P'') \wedge P'' \text{ imports } \omega\, P'$$

(7) Import is the union of the explicit package import dependency and implicit import):

$$P \text{ imports } \omega\ P' :\Leftrightarrow\ \text{exp_imports } \omega\ P' \vee P \text{ imp_imports } \omega\ P'$$

(8) Export is set of all offered nonprivate elements:

$$P \text{ exports } \omega\ E :\Leftrightarrow P \text{ offers } \omega\ E\ \wedge\ \omega\ >\ \text{private}$$

(5b) Visibility of elements across packages is extended as follows:

$$\begin{aligned} P \text{ sees } \omega\ E \quad :\Leftrightarrow \quad & P \text{ offers } \omega\ E\ \vee \\ & (\exists P' :\ P\, \text{exp_imports}\, \omega\, P'\ \wedge\ P'\, \text{exports}\, \text{public}\, E) \end{aligned}$$

We distinguish explicit import between packages and implicit import. The former is defined in UML whereas the latter needs more explanation. The implicit import comprises both, nested packages contained in a package together with their public contents, and the public contents of explicitly imported packages. Note that we used "contained in a package" and not only "owned in a packages". That means that packages themselves — being elements anyway — can be referenced in other packages. Figure 4 gives an example of some packages showing all implicit imports resulting from the nesting structure and two explicit imports.

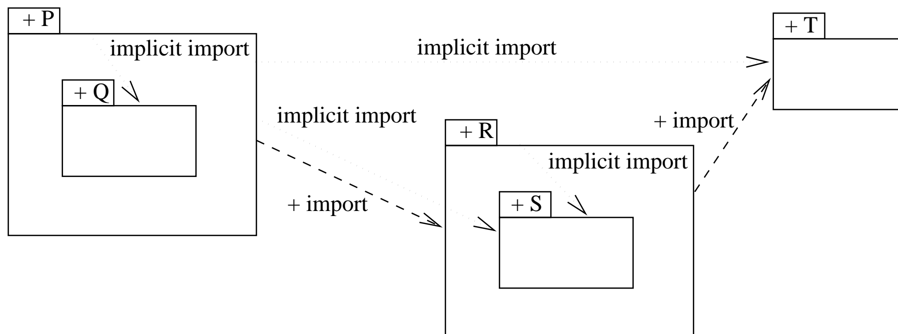

Figure 4: Explicit and implicit imports.

5 Refinement of Packages

The previous two sections introduced the "classical" modularization concepts of programming languages like Modula-2 (as defined in the ISO/IEC 10514-1 standard by Schoenhacker and Pronk (1996)), i.e. the construction of export interfaces for packages (with varying degrees of visibility inherited from C++), nesting of (local) packages, and the establishment of visible or hidden import relationships between packages. These import relationships permit access to public interface elements of server packages, only. The remaining visibility value (for interface elements) — protected — is only useful in combination with refinement (generalization) relationships between packages. The concept of refining/generalizing packages will be explained here.

Note that we give preference to the term *refinement* over *generalization* which is used in UML, because it expresses directly the relationship that we are interested in and — in most cases — helps to simplify the formulas. That means, that whenever the package `P` is source of a generalization dependency to package `Q`, `P` *refines* `Q`. The main motivation for introducing the refinement relationship between packages is that the important OO concept of inheritance should not only be available for defining single classes, the basic elements of static structure diagrams, but also for defining and refining arbitrarily complex subdiagrams.

(9) Refines* relationship is transitive closure of refinement relationship:

$$\begin{aligned} &P \text{ refines* } P' :\Leftrightarrow \\ &\quad \exists P'', \omega : P \text{ refines } \omega \ P'' \wedge \ (P'' = P' \ \vee \ P'' \text{ refines* } P') \end{aligned}$$

(10) Refinement relationship is acyclic:

$$\neg(P \text{ refines* } P)$$

(11) Public export of refining package has refined package's export:

$$P \text{ refines public } P' \ \wedge \ P' \text{ exports public } E \Rightarrow P \text{ exports public } E$$

(5c) Visibility across package boundaries is extended as follows:

$$\begin{aligned} &P \text{ sees } \omega \ E :\Leftrightarrow P \text{ offers } \omega \ E \ \vee \\ &\quad (\exists P' : P \text{ exp_imports } \omega \, P' \ \wedge \ P' \text{ exports public } E) \ \vee \\ &\quad (\exists P', \omega', \omega'' \ \geq \ \text{protected} \ : P \text{ refines } \omega' \ P' \ \wedge P' \text{ exports } \omega'' \ E \\ &\qquad\qquad \wedge \ \omega = \ \min \ (\omega', \omega'')) \end{aligned}$$

(6b) Implicit import is extended along the generalization relationship:

$$\begin{aligned} &P \text{ imp_imports } \omega \ P' :\Leftrightarrow \ P \text{ contains } \omega \ P' \ \vee \\ &\quad \exists \, P'' : \ (P \text{ contains } \omega \, P'' \ \vee \ P \text{ exp_imports } \omega \, P'' \ \vee \ P \text{ refines } \omega \, P'') \\ &\qquad \wedge \ P'' \text{ imports } \omega \ P' \end{aligned}$$

Please note that formulas (5c) and (6b) above are just an extension of formulas (5b) and (6a) of the previous section, resp. For visibility refinement relationships with different degrees of visibility are taken into account, ranging from a kind of public subtype inheritance to pure implementation inheritance. That means, that a refining package sees all public elements of the refined package as public elements (if the refinement relationship is public, too). Furthermore, a refining package sees all protected elements of the refined package as $\omega \ \geq$ protected visible elements (if the refinement relationship is ω visible, too). It is an open question, whether it makes sense to

have three different visibility cases for refinement relationships, instead of the usual distinction between interface preserving subtype inheritance and the hidden inheritance of implementations.

It is not at all difficult to come up with a precise definition of the consequences of refinement (generalization) relationships for the visibility of package elements as well as with a formal definition of the constraint "generalization relationships do not build cycles". It is far more difficult to translate the meaning of sentences like "...an instance of the subtype is substitutable for an instance of the supertype". The latter constraint cannot be defined for packages in general, but must be studied for each language of UML diagrams, separately. Such a precise definition of the term "substitutability" is not part of UML. Therefore, our formulas will only take the consequences of public refinement relationships for the visibility of (public) interface elements of related packages into account. For further details concerning the formal treatment of subtyping from an algebraic point of view the reader is referred to Breu (1991) and from a type-theoretic point of view to Palsberg and Schwartzbach (1994). It is an open question whether similar constraints have to be added for the case of non-public refinement relationships and interface elements.

6 Further Consistency Constraints

The developed formal definition of the basic UML package concept shows that on one hand, the original natural language definition seems to contain some contradictions, and, on the other hand, is not at all complete. As a consequence, we add a number of own assumptions, without knowing (until now) whether these add-ons respect the original intentions of the three amigos Booch, Jacobson, and Rumbaugh. These add-ons were kept as minimal as possible within the previous sections, thereby postponing the definition of a number of important additional constraints to this section. All these constraints, which will presented below, are — in our opinion — not a matter of debate, but should be part of UML. It is for instance useless to see a dependency without seeing its source or target package (13). Furthermore, a package should never be able to define additional server packages or generalizing packages for not locally defined packages (14). Otherwise, it would be possible to extend and modify the implementation of imported packages, thereby violating the very principle of information hiding. Last but not least it makes no sense that a package refines one of its locally defined packages (15) or that a package which is a generalization depends on one of its refining packages (16, 17).

(13) Visibility of import/refines dependency is less equal visibility of related packages:

$$P \text{ dep } \omega\ P' \Rightarrow \exists P'', \omega', \omega'' : P'' \text{ owns } \omega \text{ dep} \wedge$$
$$P'' \text{ offers } \omega'\ P \wedge P'' \text{ offers } \omega''\ P' \ \wedge\ \omega\ \leq\ \min(\omega', \omega'')$$

(14) Source of import/refines dependency belongs to owner of dependency:

$$P \text{ dep } \omega\ P' \Rightarrow P'', \omega' : P'' \text{ owns } \omega \text{ dep } \ \wedge\ P'' \text{ owns } \omega'\ P$$

(15) A package should neither import nor refine its offered (own) packages:

$$P \text{ dep } \omega\ P' \Rightarrow \neg\exists\ \omega' : P \text{ offers } \omega'\ P'$$

(16) DependsOn relationship is transitive closure of imports and refines:

$$\begin{aligned} P \text{ dependsOn } P' :\Leftrightarrow \quad \exists P'', \omega : \ & (P \text{ refines } \omega\ P'' \ \vee\ P \text{ imports } \omega\ P'') \\ & \wedge (P'' = P' \ \vee\ P'' \text{ dependsOn } P') \end{aligned}$$

(17) Package may not refine a package which is an (in-)direct client of itself:

$$P \text{ refines } P' \Rightarrow \neg(P' \text{ dependsOn } P)$$

7 Generating UML Diagram Editors

The last sections have been dedicated to giving the UML package concept a precise semantics. Why did we bother with predicate logic formulas, as one of the pros of UML is the intuitive clearness of diagrams? In fact, playing around with packages and the visibility of their elements raised many questions. It was a tedious task to find the relevant pages in Rational (1997) and many details remained unanswered.

This was our motivation to suggest predicate logic formulas. They reflect parts of the static semantics of the UML package concept given by meta model, OCL expressions, and natural language description. In order to study the rules concerning packages and visibility in more detail for different diagrams and situations, we wanted to “play” with the formulas and their effects on the diagrams. For that purpose we have translated the formulas into a *PROgrammed Graph REwriting Systems specification.*

The PROGRES language is a logic-based executable specification language (Schürr et al. (1995)). It not only permits to model the static structure of graphs with a *graph schema* very similar to the UML meta model, but furthermore, allows to describe *transformations* on these graphs with declarative graph rewrite rules. The specifications can be executed either with an interpreter, which is an integral part of the PROGRES environment, or by generating C code from the specification, which can compiled and linked together with a Tcl/Tk user interface. While the former is very useful for debugging a specification, the latter establishes a tool for working with the specification interactively.

For the UML specification the generated tool can be considered as a *UML editor* which allows the user to call typical diagram editing operations like, for instance, `AddPackage` or `AddGeneralization`. These operations have

been specified in PROGRES and perform transformations on the underlying internal graph structure ensuring the consistency according to the given graph schema (meta model). This means, the actually executed graph transformations are written in a way to respect the formulas introduced above. In figure 5 we present the PROGRES environment showing the so-called *production AddGeneralization* that inserts a new generalization dependency between two packages.

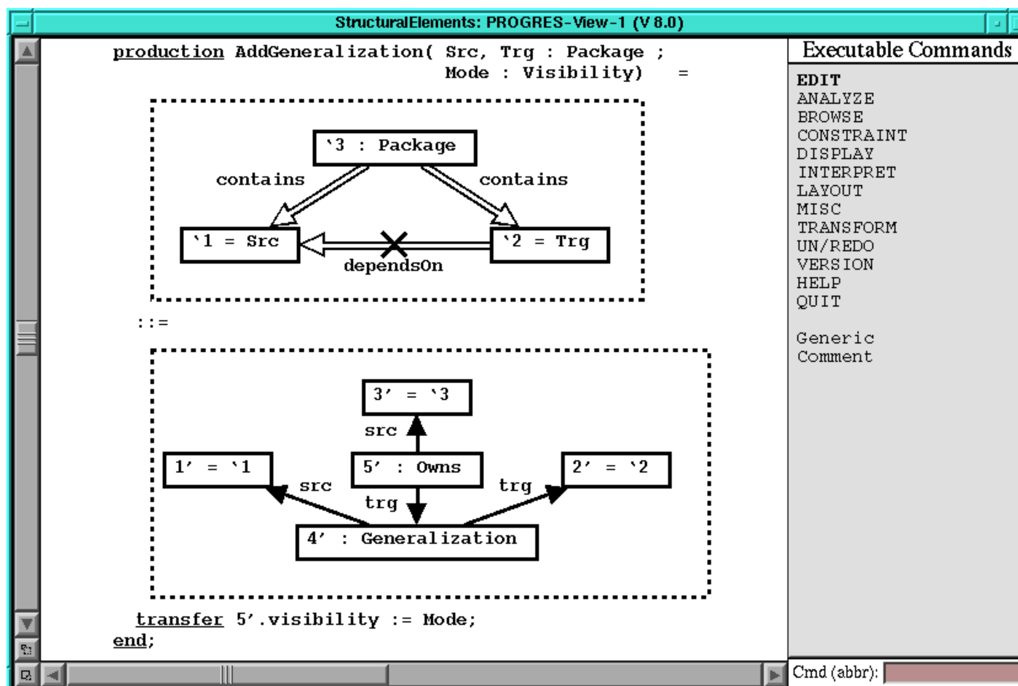

Figure 5: PROGRES production adding a generalization dependency.

A PROGRES production consists essentially of its left hand side and its right hand side. Its operational semantics is, that a match for the graph in the left hand side is searched in the underlying working graph. If a matching subgraph can be found, it is replaced by the right hand side of the rule. Nodes and edges which are present on the left hand side, but not on the right are deleted, whereas elements of the right side not appearing on the left are created. It is also possible to identically replace nodes which are identified on the left and right side. That means, the nodes are not actually deleted and reinserted, but they are retained during the application of the rule.

The example in figure 5 shows the production `AddGeneralization`. It takes two nodes as input parameters `Src, Trg` which are of type `Package`. The intention is, to establish a new generalization dependency from the package `Src` to the package `Trg`. Here, we demand that both packages are contained in a common package `‘3`. Furthermore, according to rule (17) we require

that the target of the generalization does not depend on the package which is about to become its refinement in this rule. This fact is expressed by the crossed out `dependsOn` path. This path itself is a derived graph navigation defined in the sense of rule (16).

During the execution of the rule, the three package nodes of the left side are retained (e.g. `1' = '1`) and the new `Generalization` dependency (node `4'`) is inserted and attached to its source and target. The generalization will get a visibility attribute (`transfer` part) which has also been passed as parameter `Mode`. The `Owns` node (`5'`) which is also created, denotes the `ElementOwnership` of UML's meta model. In this case, the package which contained the source and the target package of the new dependency will also be its owner.

We have seen in this example production, in which way the formulas of this paper can influence the PROGRES specification. Here, we would reject the execution of the production if no match for the left side of the rule can be found. But this is not an adequate reaction for all the rules suggested above. For example, if a user wants to use an element from another package, if the required import dependency is missing (formula (5a-c)), it is not necessary to reject the operation. Instead, we have handled this error situation with the help of an error message attribute in the used element itself. This means, that the operation is executed and an error message requests the diagram modeler to add the missing import in order to get a consistent diagram.

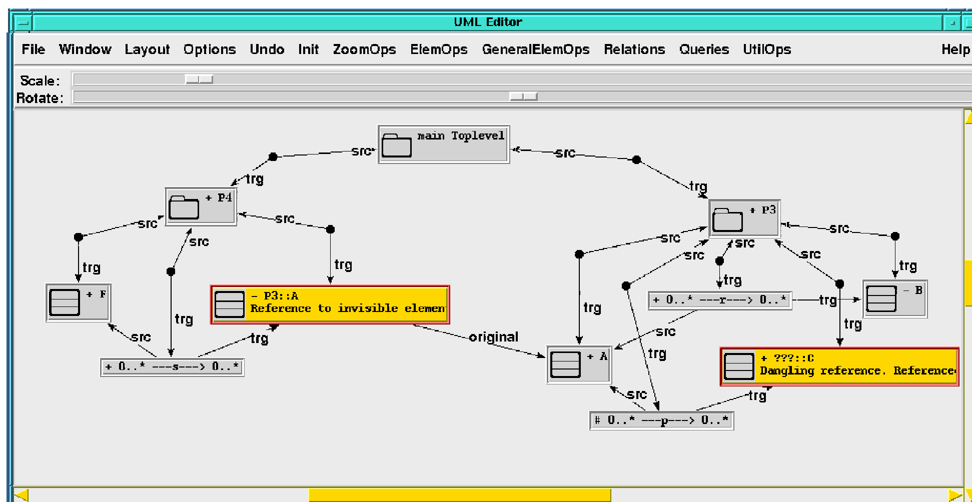

Figure 6: Session with the PROGRES generated UML editor.

In figure 6 we present a view of a UML diagram within a generated UML editor which also reveals internal dependencies, e.g. `ElementOwnership`, explicitly, in order to explain the situation sketched above. This is of course not the normal editor view. Normally, the user of the editor would only

see the UML diagram with the elements that he has inserted, not the internal dependencies. Nevertheless, the view shown here is especially useful for debugging sessions.

In this screen shot of the generated UML editor, we have three packages, two of them, namely `P3` and `P4`, are contained in package `main`. Package `P3` contains a class element `A` which is also used in package `P4`. But, the reference `P3::A` in `P4` does not "see" its origin anymore, e.g. due to the deletion of the import dependency from `P4` to `P3`. We could have forbidden the deletion because of the resulting inconsistency. But, instead, the reference node carries an error message attribute saying *"Reference to invisible element"* which demands to add the missing import in order to re-establish a consistent UML diagram.

8 Summary

This paper presented a very compact definition of the UML notation's modularization concept and suggested a number of useful extensions. These extensions compensate the obviously incompleteness of UML's natural language definition in Rational (1997) or represent additional policies for the definition of import and refinement relationships between packages. Furthermore, we gave a very short impression of translating predicate logic formulas into a PROGRES specification. This allows to generate an editor automatically, corresponding to the semantics given in the formulas. On one hand, this is very useful, if further formulas are developed. In that case the new editor is able to check the constructed diagrams against the new formulas. On the other hand, experiences with the editor can give valuable information about constellations which have not been considered adequately in the formulas to this moment.

To summarize, neither UML's module concept itself nor the considerations presented here concerning its formal definition and necessary modifications are restricted to a single object-oriented analysis and design method. On the contrary, the presented module concept may be added to other analysis, design, or specification languages or it may even build the basis for a separate module interconnection language. This is due to the fact that presented formulas make no assumptions about the semantics of basic elements in packages. It is their exclusive purpose to explain the impact of packages and relationships between packages on the visibility of package elements.

As a consequence, this paper complements the rapidly growing number of publications which have either the formal definition of module interconnection languages (architecture styles) or certain OO diagram types as their main topic. Both categories of papers assume either very simple visibility rules (compared with the visibility rules introduced here) or neglect this aspect at all due to the absence of a module concept. Examples of the first category are for instance the VDM-SL definition of the Modula-2 standard in Schoenhacker and Pronk (1996) or the Z definitions of various data flow

or event based architecture styles (Abowd and Garlan (1995), Rice and Seidman (1994)). Examples of the second category are formal definitions of (subsets of) OMT (Jonckers et al. (1996)) and Fusion (Bates et al. (1996)).

References

ABOWD G. and GARLAN D. (1995): Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology, 4(4), 319-364.*

BATES B., BRUEL J., FRANCE R., and Larrondo-Petrie M. (1996): Guidelines for Formalizing Fusion Object-Oriented Analysis Models. In *Proc. CAiSE'96*, LNCS 1080, Springer Verlag, Berlin, 222-233.

BOOCH G. (1994): Object-Oriented Analysis and Design. Benjamin Cummings Series in Object-Oriented Software Engineering. Benjamin Cummings, Redwood City, CA.

BÖRSTLER J. (1994): Programming-in-the-Large: Languages, Tools, Reusability. Dissertation (RWTH Aachen), TR UMINF 94.10, Department of Computer Science, Umeå University, Sweden (in German).

BREU R. (1991): Algebraic Specification Techniques in Object-Oriented Programming Environments. LNCS 562, Springer Verlag, Berlin.

DEREMER F. and KRON H. (1976): Programming-in-the-large versus Programming-in-the-small. *IEEE Transactions on Software Engineering, 2(2), 80-86.*

ELLIS M. and STROUSTRUP B. (1994): The Annotated C++ Reference Manual. Addison-Wesley, Reading, MA.

FIRESMITH D.G. (1993): Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach. John Wiley, New York.

JACOBSON I. (1994): Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading, MA, fourth edition.

JONCKERS V., VERSCHAEVE K., WYDAEGHE B., CUYPERS L., and HEIRBAUT J. (1996): OMT*, Bridging the Gap between Analysis and Design. In *Proc. FORTE '95*. Chapman & Hall, 39-55.

NAGL M. (1990): Software Engineering: Methodological Programming-in-the-Large. Springer Verlag (in German).

PALSBERG J. and SCHWARTZBACH M. (1994): Object-Oriented Type Systems. John Wiley, New York.

PARNAS D. (1972): A Technique for Software Module Specifications with Examples. *Communications of the ACM, 15, 330-336.*

RATIONAL ROSE SOFTWARE CORPORATION (1997): UML Semantics, Version 1.1, September 1, URL:http://www.rational.com.

RICE M. and SEIDMAN S. (1994): A Formal Model for Module Interconnection Languages. *IEEE Transactions on Software Engineering, 20(1), 88-101.*

ROBINSON P.J. (1992): Hierarchical Object-Oriented Design. Prentice Hall, Englewood Cliffs, MA.

RUMBAUGH J., BLAHA M., EDDY W., and LORENSEN W. (1991): Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ.

SCHOENHACKER M. and PRONK C. (1996): ISO/IEC 10514-1, The Modula-2 standard: Changes, Clarifications, and Additions. *ACM SIPGPLAN notices, 31(8), 84-95.*

SCHÜRR A. (1996): Logic Based Programmed Structure Rewriting Systems. *Fundamenta Informaticae, XXVI(3/4).*

SCHÜRR A. and WINTER A. (1997a): Formal Definition and Refinement of UML's Module/Package Concept. In Kilov H., Rumpe B. (eds.): Proc. ECOOP '97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques, TR TUM-I9725, Technical University Munich, also to be published in LNCS, Springer Verlag.

SCHÜRR A. and WINTER A. (1997b): Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems. Technical Report AIB 97-3, RWTH Aachen, Germany.

SCHÜRR A., WINTER A., and ZÜNDORF A. (1995): Graph Grammar Engineering with PROGRES. In Schäfer W., Botella P. (eds.): *Proc. 5th European Software Engineering Conf. (ESEC '95), LNCS 989.* Springer Verlag, Berlin, 219-234.

WIENER R. and SINCOVEC R. (1984): Software Engineering with Modula-2 and Ada. John Wiley, New York, NY.

WIRFS-BROCK R., WILKERSON B., and WIENER L. (1990): Designing Object-Oriented Software. Prentice Hall, Englewood Cliffs, NJ.