Giving Precise Semantics to Reuse in UML

Tom Mens, Carine Lucas, Patrick Steyaert
{ tommens | clucas | prsteyae }@vub.ac.be

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel
http://progwww.vub.ac.be/prog/pools/rcs/rc.html

Abstract. *In UML there is little support for reusing specifications and designs. To deal with this problem, we enhance UML with support for definition, reuse and evolution of reusable components. This is not a trivial task, since some language concepts lack a precise semantics, while the interpretation of others is simply "beyond the scope of UML". By providing our own definitions in these cases, and by making use of packages, refinements, and the built-in extension mechanisms, a precise semantics is given to reuse of UML models. Among others, this precise semantics allows us to detect reuse conflicts automatically.*

Key Words. *UML, reuse, precise semantics, reuse contracts, incremental modification.*

1. Introduction

The Unified Modelling Language version 1.1 [Rational97] has been accepted by the OMG as an industry standard for object-oriented analysis and design notation. It features a unified notation of widely accepted models such as class diagrams, use case diagrams, statecharts, sequence diagrams (or event traces) and so on.

Probably the most interesting features of UML are its built-in extension mechanisms. One possibility is to attach a *stereotype* to a modelling element. In this way, a different interpretation can be given to the corresponding element. Another way to add new functionality to UML is by making use of *constraints*. These express semantic relationships among modelling elements that specify conditions and propositions that must be maintained as true. Both stereotypes and constraints can be predefined in the language, or can be user-defined. At first sight, the availability of these extension features might indicate that adding support for truly reusable specifications and designs to UML is fairly straightforward. There are two main reasons why this is not the case.

A first problem is that UML offers only limited support for incremental modification of modelling elements. While a modelling element can be modified in many different ways, the generalisation relationship of UML only allows a conservative kind of incremental modification, where the more specific element is substitutable for the more general element. We agree that this kind of incremental modification is useful, but it most certainly is not the only kind of incremental modification imaginable.

A second problem is the lack of a precise semantics for various UML constructs. For example, the semantics of the generalisation relationship is unclear. While it is more or less obvious how generalisation between classes in a class diagram works, it is much more difficult to interpret a generalisation relationship between packages that can be arbitrarily complex. A similar observation was made by [Schürr&Winter97]. They observe that the informality of the natural language in which most of the UML semantics is expressed leads to an incomplete specification of the package definition. They try to deal with this problem by adding extra constraints on packages by means of logical formulas.

In order to solve both problems, we propose to extend UML with the *reuse contracts* formalism [Lucas97]. This allows us to give a precise semantics to reuse of specification and design models expressed in UML, which can for example be used to detect reuse conflicts automatically. In this way, the software development process can be improved.

2. Reuse of Specifications and Designs

2.1 Incremental modification

Since the main contribution of this paper is to provide support for reuse of UML models, we need to agree upon a clear definition of what reuse actually is. According to [Bassett97], *reuse is the process of adapting generalised components to various contexts of use*. Observe that this definition is sufficiently general to include evolution as well as customisation of components. A key word in the definition is the term **adapting** (or modifying). Reusing a component without the need for any modification (often referred to as black-box reuse) should not be called reuse, since it simply corresponds to **using** the component.

It is generally acknowledged that reuse should occur as early as possible in the software life-cycle, preferably at analysis and design level. Nevertheless, while there is a lot of support for reuse of implementation-level components, there is much less support for, and understanding of, reuse during the analysis and design phases. For example, in UML, which can be used to describe object-oriented analysis and design, issues such as what a reusable model is, how a reusable model can be adapted, and how models can evolve over time, are left unaddressed.

There are many ways in which a model can be reused (i.e. adapted). The simplest way is by directly editing it. This kind of reuse has the disadvantage that the original model is not available afterwards. As a result, there is no way to assess the impact of this modification on other models that depended on (parts of) the original model. Another widespread form of reuse, called *copy-and-modify reuse*, consists of taking a copy of the original model and modifying the copy. When no form of consistency is maintained between the modified model and the original one, this kind of reuse may lead to a *proliferation of different versions* of the same model.

To solve both problems, we need another kind of reuse, called *incremental modification*, where the modified model is derived from (and related to) the original model by means of an incremental modification mechanism that specifies which modifications (or adaptations) have been made to the original model. The original model however is not altered and not thrown away.

In this paper we will only deal with models describing collaborating classes. To a certain extent, the *generalisation relationship* of UML can be used to express incremental modification of collaborating class components. However, the generalisation relationship can only be used to *add* more information (because the more specific element needs to be substitutable for the more general element). This makes generalisation too restrictive for our needs. Even a simple modification like replacing an operation invocation by another one without changing the overall behaviour, is impossible with the generalisation relationship. A better solution is probably to make use of *dependency relationships*. As stated in [Rational97], a dependency indicates a semantic relationship among model elements in which a change to one element may affect or require changes to other elements. By making use of stereotypes and constraints, we will be able to customise this relationship to suit our needs.

2.2 Reuse conflicts

When a component is reused (i.e. adapted), unexpected behaviour may occur in other components that depend on it. There can be many reasons for this. The behaviour of the reused component might have changed, properties of the component that were valid before might not hold anymore, and so on. This kind of conflicts is referred to as *reuse conflicts*.

As an illustration of a reuse conflict, consider the example of an interaction diagram[1] in Figure 1 that represents part of the design for a web browser. It contains a `Browser` object and a `Document` object. In order to follow a hyperlink in the document, `resolveLink` invokes the operation `getURL` in the browser, which is used to fetch the contents of the web page pointed to by the hyperlink.

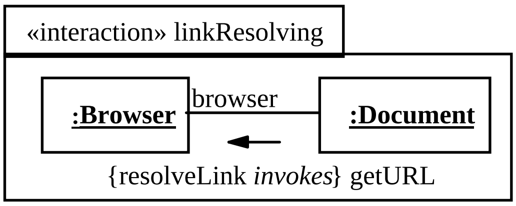

Figure 1: Collaboration diagram for web navigation

Suppose that a reuser incrementally modifies this interaction diagram to enable viewing of PDF documents (Figure 2). A PDF document only contains hyperlinks that point to places within the document itself. For this reason, the targets of these links can be retrieved by the document without needing to communicate with the browser. It suffices to incrementally modify the design of Figure 1 by removing the operation invocation from `resolveLink` to `getURL` and replacing it with an operation invocation from `resolveLink` to `gotoPage`.

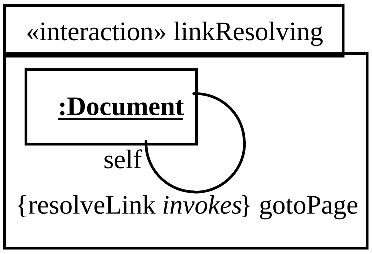

Figure 2: Incremental modification for viewing PDF documents

Now what happens if the original web browser interaction evolves by adding history functionality (Figure 3)? The idea of this modification is that, each time a hyperlink is followed through `getURL`, the URL of this link is stored somewhere through an invocation of `addURL`. This stored link can be used later on to retrieve previously followed links.

[1] The interaction diagrams used in this paper are slightly different from those proposed in UML. The motivation for this is given in section 3.4.

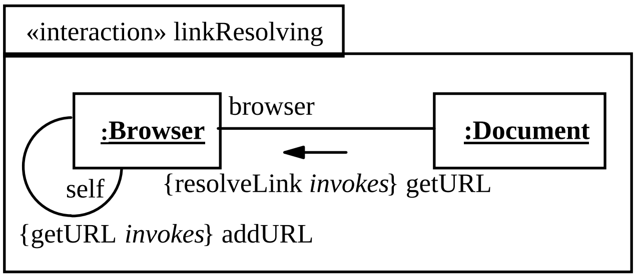

Figure 3: Evolved web navigator with history functionality

The problem that arises is that the modification for viewing PDF documents does not work together very well with the evolved web browser that exhibits history functionality. A reuse conflict occurs, since the combination of both modifications does not have the desired functionality anymore. Indeed, since link resolving is performed by the PDF document itself, the `addURL` operation in `Browser` will never be invoked, so the history will not be updated when a link is followed in the `Document`! This reuse conflict is referred to as "inconsistent operations", and will be discussed into more detail in section 5.2.

2.3 Reuse contracts

In [Steyaert&al96] and [Lucas97], the methodology of reuse contracts was proposed as a way to deal with reuse and evolution of object-oriented class hierarchies and collaborating classes. In this paper we try to translate these ideas in order to cope with reuse and evolution of UML models. Before we can do this, however, we need to give an intuitive explanation of the underlying philosophy of reuse contracts.

The essential idea behind reuse contracts is that component reuse is based on an explicit *contract* between the *provider* of a component and a *reuser* that incrementally modifies this component. The purpose of this reuse contract is to make reuse more disciplined. To achieve this goal, both the provider and the reuser have *contractual obligations*. The primary obligation of the provider is to document how the component can be reused and how the component evolves. This is specified in a so-called *provider clause*. The reuser on the other hand needs to document how the component is actually being reused. This is specified by means of a *reuser clause*.

To be able to express the specific way in which a component is reused, different kinds of reuse need to be identified. Each of these different ways is specified by a *contract type*. Possible contract types are extension, cancellation, refinement and coarsening. These types impose obligations, permissions and prohibitions onto the reuser. Contract types are fundamental to disciplined reuse, as they form the basis for detecting conflicts when provided components are reused.

3. UML

In this section we will discuss in more detail those features of UML that are needed to incorporate reuse contracts in the language.

3.1 Packages

Packages are a very important feature of UML. They provide a construct that addresses all facets of a state-of-the-art module concept. Since packages can be nested, they can be used to encapsulate arbitrarily complex components. Moreover, a nested scoping mechanism is available. Unfortunately, the informality of the UML semantics leads to an incomplete specification of the package definition [Schürr&Winter97].

In Figure 1 to Figure 3, packages were used to encapsulate the internal details of interaction diagrams. In section 3.4 we will exploit the nesting feature, and deal with more complex packages containing both a class collaboration and a set of interaction diagrams corresponding to this collaboration.

3.2 Extension mechanisms

UML contains three built-in extension mechanisms: stereotypes, constraints and tagged values. All three mechanisms can be used to enhance the expressiveness of UML in user-defined ways, and can be attached to any modelling element. From a pragmatic point of view, the possibility to add functionality to UML in a tool will depend on the extendibility of the tool and the possibility to interpret the information provided by the user-defined extensions. Currently there are very few tools supporting this capability.

A *tagged value* is a tag-value pair. The tag can be considered as a pseudo attribute. The interpretation of a tag (and its associated value) is beyond the scope of UML. User-defined semantics can be attached to it. An example is the `abstract` tag, which has a Boolean value that denotes whether an operation (or class) is abstract or not.

A *constraint*, denoted between curly braces {}, is a semantic relationship that specifies conditions and propositions that must be maintained as true. It is written explicitly in an UML diagram, preferrably in a specific constraint language like OCL, although natural language can be used as well.

A *stereotype*, denoted between guillemets «», is the most powerful way to extend UML. When a stereotype is attached to a modelling element, this indicates a distinction in usage or meaning. A stereotyped element usually specifies additional (implicit) constraints and required tagged values.

In this paper, we will make use of constraints and stereotypes to enhance UML with support for reuse. In Figure 1 we already saw an example of constraints. The constraint `{resolveLink` *`invokes`*`} getURL` indicated that `getURL` must be invoked from inside the operation `resolveLink`. As an example of a stereotype, «interaction» was attached to a package containing a kind of interaction diagram. Later in this paper we will see many more uses of the extension mechanisms to deal with reuse-contract-specific features.

3.3 Dependency relationships

In section 2.1 we already decided to use the dependency relationship of UML instead of the generalisation relationship to deal with incremental modification of reusable models. The dependency relationship is a common mechanism that can be applied to all modelling elements. It can be used to indicate situations in which changes to the supplier element may require changes to the client element. Indeed, whenever the supplier element becomes modified, the client element is potentially invalidated. How to check this invalidation, however, is outside the scope of UML! For this reason, we will need to provide a definition ourselves.

The dependency relationship is depicted by a dashed arrow from client to supplier, optionally labelled with a stereotype and a name. The UML metamodel distinguishes four kinds of dependencies: *refinement*, *usage*, *trace* and *binding*. Of these four choices, refinement seems to be the best candidate for the purpose of incremental modification. Refinement is a dependency where the clients are derived from the suppliers. The details of specifying the derivation are (again!) beyond the scope of UML, but can be indicated with user-defined constraints. Moreover, each refinement has a *mapping* association which contains a description of the mapping between the two elements. The syntax of this description is also beyond the scope of UML.

Now how can all of this be applied to the situation of reuse contracts? Both the original and the incrementally modified model will be encapsulated in a package annotated with the «provider clause» stereotype. Both packages will be connected by a refinement relationship from reuser clause to provider clause. Since we will distinguish different kinds of refinements, the refinement relationship is stereotyped with the specific kind of incremental modification that is performed («extension» in the case of Figure 4). There is a one-to-one correspondence between these stereotypes and the contract types.

To the refinement relationship, a *mapping* association is attached. This mapping will be represented by a package annotated with the «reuser clause» stereotype. Stereotyping the refinement relationship with a specific contract type automatically imposes obligations, permissions and prohibitions onto the reuser clause. For example, «extension» obliges reusers to add new items to the provided component, but prohibits the redefinition of existing items. However, adding multiple items at once is permitted. These obligations can be expressed by means of constraints attached to the stereotyped refinement.

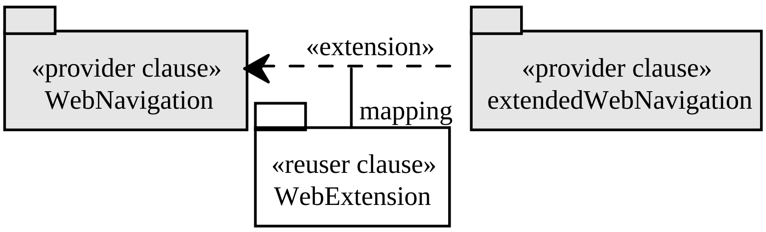

Figure 4: Incrementally extending the WebNavigation component

3.4 Design level diagrams

UML contains many different kinds of diagrams. To name but a few, we have use case diagrams, interaction diagrams, class diagrams, statecharts, deployment diagrams, and so on. Some of these are used during the analysis phase (e.g. use case diagrams), others during the design phase (e.g. collaboration diagrams), and some during the implementation phase (e.g. deployment diagrams). Since it is impossible to investigate them all in this paper, we will only look at class diagrams and collaboration diagrams. Together they can be used to describe the behaviour of collaborating classes.

The static aspects of class collaborations are represented in the metamodel of UML by the Collaboration metaclass. It basically describes the elements that participate in the collaboration (the classes or participants), their internal structure (the attributes and operations), and their relationships with other participants (the associations). Graphically, an instance of the Collaboration metaclass can be represented by using a restricted variant of class diagrams. Classes can only be connected by means of associations, not by dependency or generalisation relationships. As an extra restriction in this paper, we will only describe the class operations, not the class attributes. In other words, we work with class interfaces (denoted with the «interface» stereotype). Except for the association roles and navigability information, all other graphical adornments are suppressed. Moreover, navigation arrows are only shown if the association is navigable in one direction only.

The dynamic aspects of a class collaboration are represented in the UML metamodel by the Interaction metaclass. It describes the object interaction or message sending behaviour. Graphically, instances of the Interaction metaclass are represented by means of *interaction diagrams*. In UML, two equivalent kinds of interaction diagrams are distinguished: sequence diagrams and collaboration diagrams. In this paper, we will use a restricted form of collaboration diagrams. Message expressions will only contain the name of the message that is sent between two objects. Furthermore, we will not attach any adornments like «parameter», «local» or «global» to the associations. The most important difference with UML is that we choose not to attach a sequence number to each message. Instead, we prefer to express directly which operation invokes which other operation. For example, `{resolveLink` *`invokes`*`}` `getURL` indicates that the `getURL` operation is invoked by the `resolveLink` operation. Our approach is less restrictive than UML's, since it allows us to omit ordering information when the order of message sends is irrelevant.

In the future we will investigate how the imposed restrictions on collaborations can be loosened.

While we have already seen in Figure 1 to Figure 3 that the interaction diagrams are encapsulated in a package annotated with the «interaction» stereotype, the class diagrams representing the static aspects of the collaboration will be encapsulated in a package annotated with the «collaboration» stereotype. A «collaboration» package together with a set of «interaction» packages will be used to describe the complete class collaboration (static aspects as well as dynamic aspects). This is achieved by encapsulating all the necessary diagrams in a package annotated with the «provider clause» stereotype.

In Figure 5, a class collaboration for the web navigation example is shown. The «collaboration» package shows the important class interfaces of each participant and how they are related. Two «interaction» packages are used to express the essential behaviour: `linkResolving` and `mouseClicking`. The latter specifies what happens when the `Browser` receives a `handleClick` message.

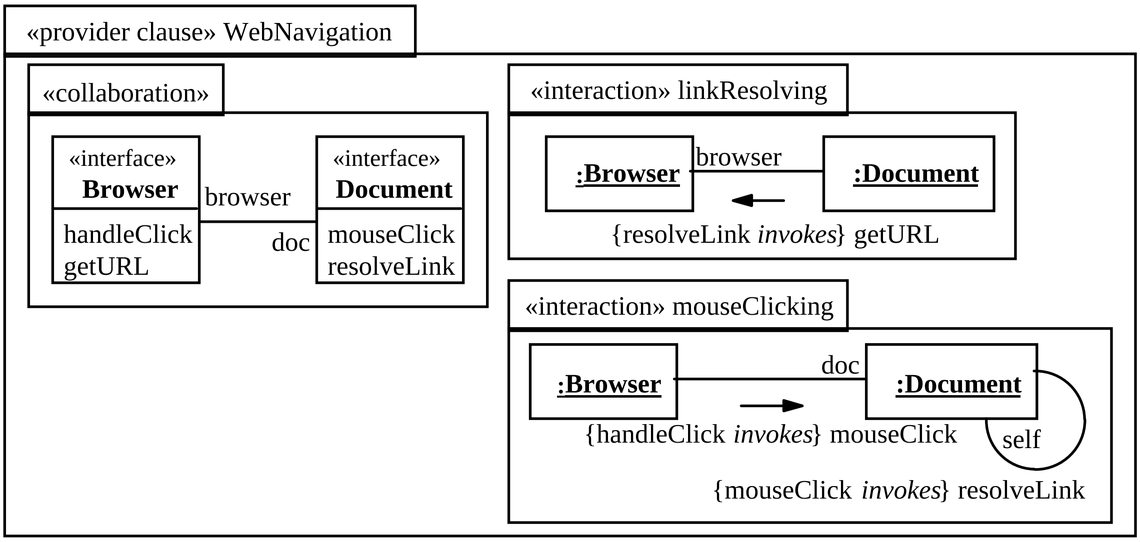

Figure 5: Package encapsulating a class collaboration

4. Formalising Reuse of UML Diagrams

In this section we try to formally define those concepts that are needed to cope with reuse of UML models. Although some of these concepts are already available in UML, we will provide an alternative definition here, in order to be consistent with the other definitions in this paper. In the near future however, we will translate all our definitions to a form that is consistent with the definitions given in the UML semantics [Rational97].

4.1 Formal definitions

First we need to formally define collaborations, interactions and how they are related. More specifically, an interaction will always be associated to a collaboration.

Definition:

A **collaboration** (P,A) is a couple consisting of a set P of participants and a set A of navigable associations. A **participant** $p \in P$ is a couple (n,O) consisting of a unique name n and a set O of operation signatures. A **navigable association** $a \in A$ is a triple (n,s,t) consisting of a name n, a source participant $s \in P$ and a target participant $t \in P$. Moreover, any two navigable associations that have the same source participant must have different names.

Notation:

If $S=(P,A)$ is a static structure we write $participants(S)=P$ and $associations(S)=A$. If $p=(n,O)$ is a participant we write $name(p)=n$ and $operations(p)=O$. If $a=(n,s,t)$ is a navigable association we write $name(a)=n$, $source(a)=s$ and $target(a)=t$.

Definition:

An **interaction** (n,D) **associated to a collaboration** S is a couple consisting of a name n and a set D of operation invocations. An **operation invocation** $d \in D$ is a triple (a,o,p) such that $a \in associations(S)$ and o, p are operation signatures.

Notation:

If $I=(n,D)$ is an interaction we write $name(I)=n$ and $invocations(I)=D$. If $d=(a,o,p)$ is an operation invocation we write $association(d)=a$, $caller(d)=o$, $callee(d)=p$, $source(d) = source(a)$ and $target(d) = target(a)$.

Unlike collaborations, interactions need to be distinguished by their name, since many different interactions can be associated to the same static structure, as we have already illustrated in Figure 5.

4.2 Maintaining the consistency of provider clauses

In Figure 5 we showed what provider clauses look like. They always contain a collaboration package and a set of interaction packages. Obviously, since provider clauses describe reusable components, the interactions in a provider clause must be **consistent** with the collaboration. The interactions can only contain instances of classes that are present in the collaboration, and operation invocations from one object to another must satisfy the constraints that the operation signatures are present in the corresponding class interfaces in the collaboration. In UML, these consistency constraints are given by means of well-formedness rules in OCL.

Definition:

An interaction I associated to static structure S is **consistent with** S if $\forall\ d \in\ invocations(I)$:

$$caller(d) \in operations(source(d)),\ callee(d) \in operations(target(d))$$

Consistency between interactions and the collaboration alone is not enough. The different interactions in a provider clause need to be mutually consistent as well. This constraint is not formally defined in UML, but can be added in a fairly straightforward way. For the moment, we have taken the easy solution where all interactions describe independent behaviour. This can be guaranteed by prohibiting each operation from playing the role of caller in more than one interaction. In other words, all messages sent by the same operation must be shown in the same interaction. Since this is a rather strong restriction, we plan to weaken it in the future.

Definition:

Two interactions I_1 and I_2 associated to the same static structure S are **consistent** if

$$name(I_1) \neq name(I_2) \text{ and}$$

$$\forall\ d_1 \in\ invocations(I_1){:}\ \forall\ d_2 \in\ invocations(I_2){:}\ caller(d_1) \neq caller(d_2)$$

Now that we have defined all these consistency rules, we can define the notion of provider clause formally:

Definition:

A **contract clause** $(n,S,ISet)$ is a triple consisting of a name n, a static structure S, and a set of consistent interactions $ISet$ associated to this static structure S. This contract clause is **consistent** if $\forall\ I \in\ ISet{:}\ I$ is consistent with S. A **provider clause** is a consistent contract clause.

Notation:

If $C=(n,S,ISet)$ is a contract clause we write $name(C)=n$, $structure(C)=S$ and $interactions(C)=ISet$. If $p \in participants(structure(C))$, then we also write $p \in participants(C)$. If $d \in invocations(I)$, with $I \in interactions(C)$, then we also write $d \in invocations(C)$.

The provider clause given in Figure 5 satisfies the above consistency requirements. First of all, the `mouseClicking` interaction and the `linkResolving` interaction are consistent, since `mouseClicking` only describes the behaviour of `handleClick` and `mouseClick`, while `linkResolving` only specifies the behaviour of `resolveLink`. Secondly, both interactions are consistent with the collaboration, since all the objects, associations and operation signatures used in the interactions are defined in the collaboration.

Although the provider clauses defined here were very simple, a lot of work had to be done. If we try to incorporate more information in the provider clause, for example by introducing a statechart to specify the intra-class behaviour, this problem will only become worse. We will have to define how this new diagram is related to the other diagrams, and provide new consistency rules to cope with changes in the new diagram.

4.3 Reuser clauses

Besides provider clauses, reuse contracts also contain reuser clauses. Reuser clauses describe the incremental modifications that are made to the provider clauses. These modifications can be made by adding or removing information in different places in the provider clause. The easiest way to express this in UML is probably by using a tag-value pair with tag `modification` that can have three values: `none`, `added` and `removed`. For example, an operation signature that is annotated with `{modification=removed}` will be removed in the reused provider clause. In the rest of this paper we will use the abbreviation `{removed}` and `{added}`.

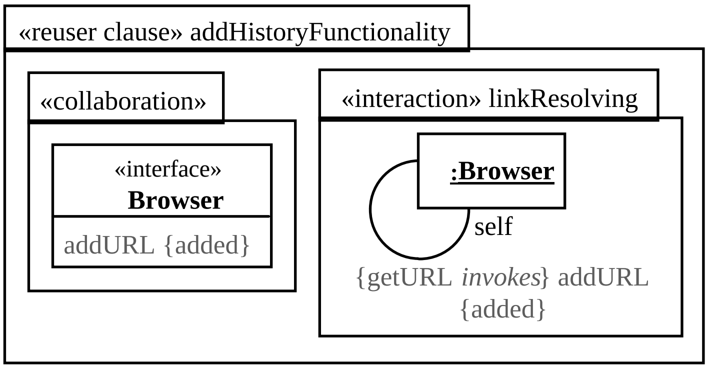

Figure 6: Reuser clause example

Since reuser clauses enumerate the modifications only, they contain only partial information. Hence, the interactions in a reuser clause are not necessarily consistent.

Definition: A **reuser clause** *C* is a contract clause (that is not necessarily consistent) where each element is optionally stereotyped with a modification tag `{added}` or `{removed}`.

4.4 Reuse contracts and contract types

Now that we have defined provider clauses and reuser clauses, we can also define reuse contracts as a mechanism that binds them together. This agreement is characterised by a contract type.

Definition:

A **reuse contract** (N,C_P,C_R,C_M,T) is a five-tuple consisting of a name N, a provider clause C_P, a reuser clause C_R, a modified provider clause C_M and a contract type T. The **contract type** T is an annotation that describes how the original provider clause is incrementally modified.

Notation:

If $R=(C_P,C_R,C_M,T)$ is a reuse contract we write $provider(R)=C_P$, $reuser(R)=C_R$, $result(R)=C_M$ and $type(R)=T$.

Note that the above definition of reuse contract is somewhat redundant. If desired, the modified provider clause C_M can be omitted, since it can be generated automatically from the provider clause C_P and the reuser clause C_R. Alternatively, the reuser clause C_R can be omitted, since it can be calculated automatically as the difference between the modified and original provider clause.

With these reuse contracts it is possible to detect reuse conflicts. The kinds of conflict that can occur depend on the particular contract type that is used. Many different contract types can be distinguished. In Table 1, the most elementary contract types are enumerated. They are subdivided into *participant* contract types and *context* contract types. The reason for this is that contract clauses provide information on two levels: the *context level* and the *participant level*. The context level is part of the collaboration diagram and describes participants (i.e. class interfaces) and their associations. The participant level can be found partially in the collaboration and partially in the interactions and includes the operation signatures and the operation invocations.

Basic Contract Type	Meaning
«participant extension»	adds new operation signatures to participants
«participant cancellation»	removes operations signatures from participants
«participant refinement»	adds new operation invocations to a participant
«participant coarsening»	removes operation invocations from a participant
«context extension»	adds new participants to the context
«context cancellation»	removes participants from the context
«context refinement»	adds new associations to the context
«context coarsening»	removes associations from the context

Table 1: Basic Contract Types

Both at participant and at context level an *extension* adds the most elementary modelling elements: respectively, operation signatures and participants. *Refinement* introduces new dependencies between these elementary modelling elements: «participant refinement» connects operations by adding operation invocations, «context refinement» connects participants by adding navigable associations. *Cancellation* and *coarsening* are the inverses of extension and refinement, respectively. Together these contract types can model all possible modifications of contract clauses. Note, however, that these eight basic types are only sufficient to model all changes to our *current* model. When new kinds of diagrams or modelling elements are added to the provider clause, extra contract types will be required.

4.5 Embedding reuse contracts in UML

Now how can these contract types be embedded in UML? Each contract type corresponds to a stereotyped refinement relationship between the original and the modified provider clause. For each such stereotyped refinement, the exact mapping between both provider clauses is specified in a *mapping* association of which the syntax is outside the scope of UML. This allows us to give our own definition to this mapping in the form of *reuser clauses*. Besides being syntactically correct, the mapping (i.e. the reuser clause) should also satisfy some semantic constraints. These constraints depend on the contract type that is being used and will be attached to the stereotype corresponding to the contract type.

Each mapping is fully characterised by two constraints. The first constraint, called the "*mapping applicability*", describes what conditions a reuser clause must satisfy in order to be applicable to a particular provider clause. The second constraint, called the "*mapping validation*", checks if the reuser clause is valid, i.e. if the modified provider clause actually is the result of applying the reuser clause to the original provider clause.

For a full definition of all basic contract types (using a different notation and terminology) we refer to [Lucas97]. We will only give the formal definition of one specific basic contract type as an example, namely «participant extension». Also note that we use standard mathematical set notation to express the applicability and validation constraints. To be fully compatible with UML, it is probably better to express these constraints in OCL.

Mapping syntax. For a participant extension, the only information needed in the reuser clause are the added operation signatures and their corresponding participants. No interactions are needed, so these will not be mentioned in the reuser clause.

Definition:

A **reuser clause** C_R **of type «participant extension»** is a reuser clause with $interactions(C_R)=\emptyset$. Moreover, $\forall\ p\in participants(C_R)$: $operations(p)\neq\emptyset$, and to each such operation signature a modification tag with value `added` is attached.

Mapping applicability constraint. For a reuser clause of type participant extension two applicability conditions are needed. First, operation signatures can only be added to existing participants. Second, only new operation signatures can be added to these participants. The reason for this is that we want to model the addition of new functionality, not the alteration of existing functionality.

Definition:

A reuser clause C_R **is applicable to** a provider clause C_P **with respect to the type «participant extension»** if

(1) C_R is a reuser clause of type «participant extension»;

(2) $\forall\ p\in participants(C_R)$: $\exists\ q\in participants(C_P)$ such that

$name(p)=name(q)$ and $operations(p)\cap operations(q)=\emptyset$.

Mapping validation constraint. If the modified provider clause is obtained by augmenting the participants in the original provider clause with the operation signatures in the reuser clause, then we have a valid «participant extension» reuser clause.

Definition:

Let C_P be the original provider clause, C_M the modified provider clause and C_R the reuser clause corresponding to contract type **«participant extension»**. C_R is a valid reuser clause if:

(1) C_R is applicable to C_P with respect to type «participant extension»

(2) $interactions(C_M) = interactions(C_P)$

(3) $associations(C_M) = associations(C_P)$

(4) $\forall\ q\in participants(C_P)$: $\exists\ p\in participant(C_M)$ with $name(p)=name(q)$ and such that:

if $\exists\ r\in participants(C_R)$ with $name(r)=name(p)$

then $operations(p)=operations(q)\cup operations(r)$

else $operation(p)=operations(q)$

5. Detecting Reuse Conflicts

Now that we have formally defined reuse contracts, and we have seen how they can be embedded in UML, we illustrate how they can actually be helpful in improving the software development process. This will be done by showing how reuse contracts allow (some) reuse conflicts to be detected automatically.

5.1 Kinds of Reuse Conflicts

Reuse contracts aim at maintaining a maximum degree of consistency between reusable components and the systems in which they are reused. More specifically, reuse contracts provide feedback on possible conflicts that occur during reuse. In general, reuse conflicts can occur when two independent modifications are made to the same model. To detect whether there is a conflict, we need to investigate how these modifications can be reused together. This section inventorises a number of reuse conflicts that can occur and describes how reuse contracts aid in detecting them.

A first set of problems that can occur are *interface conflicts*, for example, conflicts of operation signatures or participant names. It is possible that two modifications separately introduce an extra operation or participant with the same name to model similar behaviour, but also that they aimed at modelling something completely different. Therefore, such a potential conflict should be noted, to allow the user to decide how to solve this conflict.

The inverse kind of interface conflicts can occur when one modification removes some item from its interface, while another modification relies on this item. We call these conflicts *dangling reference conflicts*. Again, this kind of conflict can occur in different parts of the interface.

While the above conflicts are more or less simple, other conflicts are much more subtle. They do not necessarily make a model inconsistent, but may result in a model that does not express the expected behaviour. One example is the *inconsistent operations* conflict that was already discussed informally in section 2.2. A related conflict is *operation capture* which occurs when one modification adapts a certain operation relying on the fact that this will only have a local effect, while another modification adds an invocation of this operation, thus causing the change to have a much larger influence. We then say that the first operation is captured by the second.

We always try to detect conflicts between two reuse contracts with the same provider clause, as this models two modifications of the same component. There are different approaches possible to detect the conflicts. In most cases, it is possible to detect a conflict by comparing the two contract types and reuser clauses. An example of this is the inconsistent operations conflict, that will be elaborated upon below. Sometimes, however, the information provided by the contract types and reuser clauses does not suffice. In those situations, the original provider clause or even the modified provider clause needs to be consulted. This is the case for conflicts that involve the transitive closure of operations invocations. A discussion of these conflicts is however outside the scope of this paper. For a more extensive discussion on possible conflicts, we refer to [Lucas97].

5.2 Inconsistent Operations Conflict

We will now look into more detail at the inconsistent operations conflict as an example. Inconsistent operations appear when operation invocations are removed. This can only be achieved by a «participant coarsening». For the conflict to occur, the other reuser should change the operation from which the invocation was removed, so it can only be a «participant refinement» or a «participant coarsening». This leads to the following rule.

<u>Rule</u>: Let C_1 be a reuser clause of type «participant coarsening» and C_2 a reuser clause of type «participant refinement» or «participant coarsening». Two operation signatures *m* and *n* become **inconsistent** in participant *p* if

(1) $\exists\ d_1 \in invocations(C_1)$ such that $caller(d_1)=m$, $callee(d_1)=n$ and $target(d_1)=p$;

(2) $\exists\ d_2 \in invocations(C_2)$ such that $caller(d_2)=n$ and $source(d_2)=p$.

To resolve this conflict the operation *m* needs to be adapted in one of the two reuser clauses in order to get its behaviour consistent again. Note that rules as the one above only signal, for example, that two operations have become inconsistent. Whether this is a problem or not will depend on the situation, more particularly on the reasons why the invocation was removed. Note also that the order in which the reuser clauses are regarded in the rule does not matter.

To show how this rule works in practice, consider the `addHistoryFunctionality` reuser clause of Figure 6. To obtain a web browser with history functionality, the `WebNavigation` provider clause is incrementally modified by extending the `Browser` participant with a new operation signature `addURL`, and by refining the `Browser` participant with an extra operation invocation from `getURL` to `addURL`. The web browser can also be incrementally modified for viewing PDF documents. This is done by first removing the operation invocation from `resolveLink` to `getURL` (by means of a participant coarsening), and then adding an operation invocation from `resolveLink` to `gotoPage` (by means of a participant refinement).

If we apply the rule for detecting an inconsistent operation to both incremental modifications of the `WebNavigation` provider clause, we actually find a conflict. Indeed, according to the rule, `resolveLink` becomes inconsistent with `getURL` in participant `Browser` (see Figure 7). When we inspect this conflict more closely, we see that links that are followed inside a PDF document are not automatically added to the history, since `addURL` is not invoked anymore. Whether or not this is a real problem is up to the designer to decide. If desired, the conflict can be resolved by letting `resolveLink` call `addURL` directly.

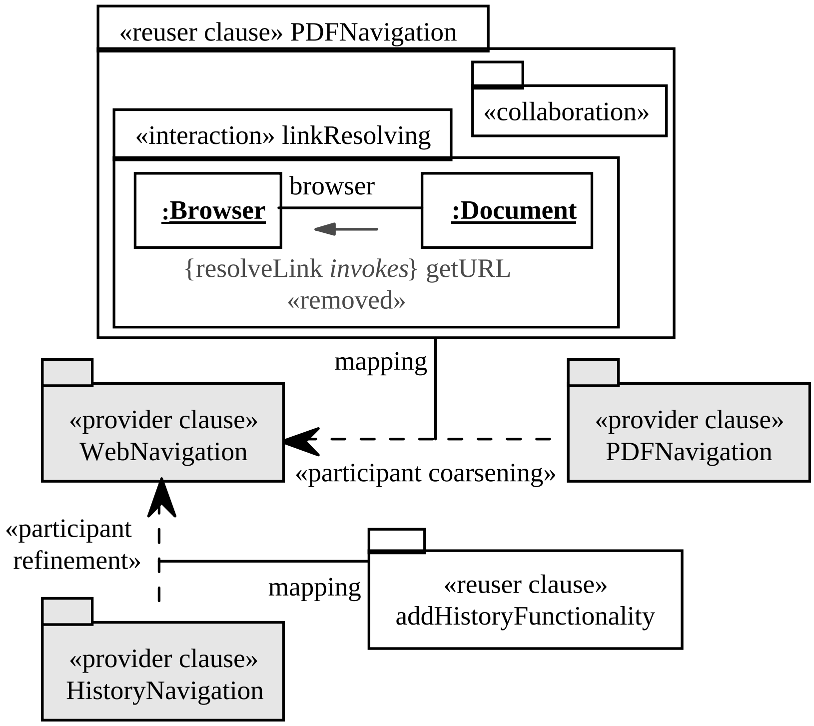

Figure 7: Detecting an inconsistent operations conflict

Above we gave an example of a rule to detect one particular conflict. A whole range of rules can be set up to detect different conflicts. As the conflicts that can possibly occur are dependent of the contract type, tables can be set up where both the rows and columns represent contract types and the fields specify what conflicts can possibly occur for a certain combination of types. This table can be filled in by comparing all contract types two by two, and by determining whether they can interact in an undesired way. Using these tables it becomes possible to detect conflicts automatically.

Obviously, when a new kind of diagram is added to the contract clauses, a whole range of new conflicts can arise. This means that the table of conflicts needs to be adapted accordingly, and rules to detect these new conflicts will have to be defined.

5.3 Scaling Up Reuse Contracts

Although the ideas presented until now describe the essence of the reuse contract formalism, they are still too primitive to be practically useful in more complex situations. To cope with this problem, we need to find out how the ideas can be scaled up.

The contract clauses can be scaled up by augmenting them with new kinds of diagrams or modelling elements that express different information. A second way is by nesting contract clauses inside each other (making use of the nesting facility of packages), so that we can deal with reuse at different levels of abstraction.

Concerning contract types, it is obvious that the basic ones presented in Table 1 are too primitive to be practically useful. In practice, a sequence of contract types will be used to build up a more complex *combined contract type*. A very natural way deal with combined contract types in UML is by making use of the ability of dependencies to serve as a container for a group of subdependencies. Subdependencies will then serve as basic contract types, while their container dependency corresponds to a combined contract type. Of course, this process can be repeated to create even more complex contract types.

An example of a frequently used combined contract type is «participant factorisation». It can be defined as a sequence of participant extensions, participant coarsenings and participant refinements. Participant factorisation can be used to factor out some part of the behaviour of one or more operations into an intermediary operation. As a result of this, the operations that were originally invoked will still be invoked after the factorisation, but through an extra indirection. In contrast to the basic contract types we have seen until now, participant factorisation does not really alter the behaviour of a system. It just makes the design more modular, and often better adaptable.

Now the natural question arises how the checking of reuse conflicts behaves when combined contract types are considered. As a first approximation, conflicts caused by combined contract types can be detected by considering the basic contract types from which they are made up two by two. This would however lead to the detection of too many conflicts, as subsequent reuser clauses in a sequence may cause local conflicts to be annihilated. For example, if a «participant extension» with a certain operation signature is followed by a «participant cancellation» of the same signature, the conflicts caused by the extension should not be considered. Therefore, before detecting reuse conflicts the sequence should first be transformed so that each reuser clause is independent – with respect to the possible conflicts – of the preceding ones. Under these restrictions, our approach is scalable to combined contract types.

More importantly, explicitly using combined contract types allows the reuser to give extra feedback on which conflicts are real and which probably aren’t. A good example can be found with «participant factorisation». Inconsistent operations occur when an invocation of operation *m* is *removed* by one reuse contract and this same operation *m* is refined or coarsened by another reuse contract. Since a factorisation involves a «participant coarsening», an inconsistent operations conflict can appear after a «participant factorisation». Upon closer examination however, we see that there is an intuitive difference between a stand-alone «participant coarsening» and a «participant coarsening» that is part of a «participant factorisation». When an operation invocation is removed during a «participant factorisation», this invocation always added to another operation with the net result that it remains in the transitive closure of the operation it was originally removed from. Therefore, one does not get true inconsistent operations. In other words, by explicitly declaring the «participant coarsening» to be part of the «participant factorisation», the user expresses that the inconsistent operations conflict can be neglected. This result can be translated into the following rule:

Inconsistent Operations Annihilation: The conflict of inconsistent operations that can be caused through a «participant coarsening» is neglected when this «participant coarsening» is part of a «participant factorisation».

6. Conclusion

It is generally acknowledged that reuse should happen preferably as early as possible in the software life-cycle. Nevertheless, there is little or poor methodology and tool support for reusable specifications and designs. To deal with this problem, we enhanced UML with support for definition, reuse and evolution of reusable components. More specifically, we have shown how to incorporate the reuse contracts formalism into UML.

In order to do this, we needed to cope with two problems. First, the informality of the natural language in which part of the UML semantics is expressed leads to an incomplete specification of several language concepts. Secondly, the interpretation of some language features is outside the scope of UML, and needs to be provided by the user.

Despite these problems, extending UML with reuse functionality was possible, thanks to three powerful language features. First of all, the package mechanism enabled us to encapsulate arbitrarily complex reusable components. Second, the refinement relationship was a readily available mechanism to express incremental modification of these components. Last but not least, the built-in extension mechanisms of UML allowed us to fine-tune the employed modelling elements to our specific needs. Moreover, scalability of the approach was ensured thanks to the nesting capability of packages, and the ability of dependencies to serve as a container for a group of subdependencies!

In this way, a precise semantics was given to reuse of UML models. This precise semantics actually gives an added value to UML, since reuse conflicts can be detected automatically. Other benefits are the ability to maintain the consistency between different diagrams, and the ability to deal with propagation of changes and proliferation of versions. It is obvious that all these benefits will help in improving the software development process.

7. Future Work

In this paper we used mathematical set notation to specify constraints like consistency rules on provider clauses, and applicability rules and validation checks on reuser clauses. Maybe a better idea is to use the OCL, since this language is also used in the UML semantics to express constraints on the different modelling elements [Rational97].

In order for UML to fully support reuse of specifications and designs, the reuse contract formalism discussed in this paper still needs to be elaborated upon in several ways.

The proposed model should be enhanced further to support the full functionality of collaborations. A challenging task will be to incorporate order on operation invocations in interaction diagrams. The main problem here is that it is still unclear how an interaction diagram can be incrementally modified in a meaningful way. In the UML semantics, no details are given about generalisation or refinement of interaction diagrams.

In order to be a full extension of UML, reuse contracts also need to be incorporated in use case diagrams, sequence diagrams, state-transition diagrams, and so on. Again the problem arises that it is unclear what the useful incremental modifications of these diagrams are. Applying reuse contracts to sequence diagrams should not be too difficult once we know how to deal with incremental modification of collaboration diagrams. An early attempt to incorporate reuse contracts in state-transition diagrams has already been made in [Mens&Steyaert97]. For use case diagrams however, we still need to perform a lot of work.

Another important step is to investigate how all the different UML diagrams are related to one another, to be able to maintain the consistency of the model. If a modification is made to one diagram, what is the impact on other diagrams? Related to this issue is the problem of transition between software phases. Describing the dependencies between models used in different phases of the software life-cycle (e.g. design phase and implementation phase) by means of reuse contracts leads to a better tractability between the software phases. In this way, the impact of making a change in an earlier phase on models in a later phase can be assessed. Conversely, we will get a better idea about how modifying models in a later phase makes them drift away from models used in earlier phases.

8. References

[Bassett97] Paul G. Bassett: "Framing Software Reuse: Lessons From the Real World", Yourdon Press Computing Series, ISBN 0-13-327859-X, Prentice Hall, 1997.

[Lucas97] Carine Lucas: "Documenting Reuse and Evolution with Reuse Contracts", PhD Dissertation, Programming Technology Lab, Vrije Universiteit Brussel, September 1997.

[Mens&Steyaert97] Tom Mens and Patrick Steyaert: "Incremental Design of Layered State Diagrams", Technical Report vub-prog-tr-97-04, Vrije Universiteit Brussel, 1997.

[Rational97] Rational Software Corporation: "Unified Modelling Language 1.1 Document Set", http://www.rational.com, September 1997.

[Schürr&Winter97] Andy Schürr and Andreas J. Winter: "Formal Definition and Refinement of UML's Module/Package Concept", ECOOP '97 Workshop on Precise Semantics for Object-Oriented Modelling Techniques, ECOOP '97 Workshop Reader, LNCS 1357, pp. 211-215, Springer-Verlag, 1998. Also available as Technical Report TUM-I9725, pp. 141-147, Technische Universität München, 1997.

[Steyaert&al96] Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt: "Reuse Contracts: Managing the Evolution of Reusable Assets", Proceedings of OOPSLA '96, ACM SIGPLAN Notices, 31(10), pp. 268-286, ACM Press, 1996.