Giving Precise Semantics to Reuse and Evolution in UML

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

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel

Abstract. *Next to the fact that UML is becoming an industry standard, probably one of its most salient features is its built-in extension mechanisms, namely stereotypes and constraints. This should make it easy to extend UML notation with new functionality. Unfortunately, the lack of a precise semantics for UML is one of the main inhibitors of its extendibility. Many difficulties need to be overcome in order to add reuse and evolution features to UML.*

Key Words. *UML, evolution, reuse, precise semantics, reuse contracts*

1. Introduction

The Unified Modelling Language (UML 1.1, [Rational97]) has recently been accepted by the OMG as an industry standard. 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.

One of the most interesting features of UML is its built-in extension mechanisms. One possibility is to attach a *stereotype*, denoted between guillemets («…»), 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 are semantic relationships among modelling elements that specify conditions and propositions that must be maintained as true. Constraints are denoted between curly braces ({…}). Both stereotypes and constraints can be predefined in the language, or can be user-defined.

Despite these extendibility features, the lack of a precise semantics for UML is the main inhibitor for supporting truly reusable specifications and designs. For example, 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. Another problem is that 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 solve this problem by adding extra constraints on packages by means of logical formulas.

Another problem with UML is that it doesn't formally specify how different kinds of UML diagrams are related. This is however a very important question in the context of evolution. For example, when a class in a class diagram is modified during evolution, this can have an impact on the statechart that specifies the internal behaviour of the class, but also on the collaboration diagrams corresponding that describe the interaction with other classes, or even on the analysis-level use case diagram that was originally used to create the design-level class diagram. Actually, this corresponds to the problem of change propagation. Making a change in one diagram can have an impact on a lot of other diagrams, where changes may need to be made as well.

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 and evolution of specification and designs expressed in UML. As a result of this, it will become much easier to maintain the consistency between the different modelling elements. Moreover, a precise semantics for evolution allows us to detect evolution conflicts automatically. In this way, the software development process can be improved.

2. Reuse and Evolution of Specifications and Designs

2.1. Incremental modification of analysis and design models

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 and evolution 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 component is, and how reusable components can evolve over time, are left unaddressed.

In this paper we will only deal with reusable components describing collaborating classes. There are many ways in which such a reusable component can be modified during evolution. The simplest way is to modify a component by directly editing it. This is often referred to as copy-and-paste reuse. This kind of reuse has the disadvantage that it is unclear how the component is adapted, and that the original component is not available afterwards. So, in general, an *incremental* modification mechanism is preferred. With incremental modification, a modified component is obtained by composing an existing component with a component modifier through some modification mechanism.

In UML, a generalisation relationship can be used to express incremental modification of collaborating class components. However, this kind of relationship has the drawback that it 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 a lot of purposes. Even a simple modification like replacing an operation dependency by another one without changing the overall behaviour, is impossible with the generalisation relationship.

2.2 Evolution conflicts

When a component evolves, unexpected behaviour may occur in other components that depend on it. There can be many reasons for this. The behaviour of the evolved 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 *evolution conflicts*.

As an illustration of an evolution conflict, consider the example of a collaboration diagram[1] that represents part of the design for a web browser (Figure 1). 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.

---

[1] The collaboration diagrams used in Figures 1 to 3 are slightly different from those proposed in UML.

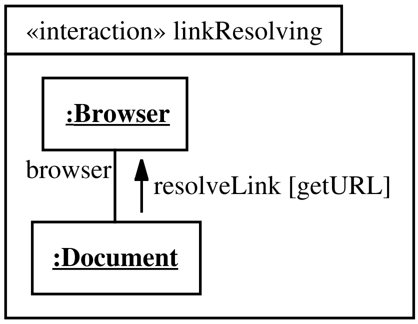

Figure 1: Collaboration diagram for web navigation

Suppose that another component incrementally modifies this collaboration 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`.

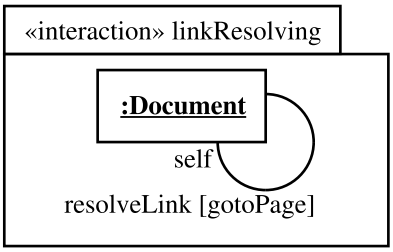

Figure 2: Modification for viewing PDF documents

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.

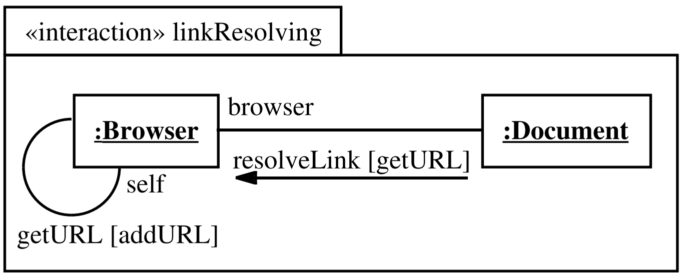

Figure 3: Evolved web navigator with history functionality

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. An evolution conflict occurs, since the evolved web browser 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`!

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 allow reuse and evolution of UML diagrams. 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 reuse and evolution of components is based on an explicit *contract* between the *provider* of a component and a *reuser* that 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 evolve.

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 [Schürr&Winter97]. 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.

In Figures 1 to 3, packages were used to encapsulate the internal details of interaction diagrams. In section 3.4 we will see how to deal with more complex packages containing both a class diagram and a set of interaction diagrams.

3.2 Extension mechanisms

As already mentioned in the introduction, UML contains two built-in extension mechanisms: stereotypes and constraints. Both mechanisms can be used to enhance the expressiveness of UML in user-defined ways, and can be attached to any modelling element. The main difference is that constraints are written explicitly in the UML diagrams, usually in a specific constraint language (like OCL), while stereotypes correspond to "hidden" constraints. When a stereotype is attached to a modelling element, this indicates a usage distinction. A stereotyped element may have additional (implicit) constraints on it.

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 parse the information provided by the extension mechanisms. In other words, there must at least be a way to add user-defined stereotypes and constraints, and to specify how these constraints should be interpreted.

In this paper, we will only make use of stereotypes to extend UML with reuse and evolution functionality. We already saw one example in Figures 1 to 3, where the stereotype «interaction» was attached to a package containing a kind of interaction diagram. In the rest of this paper, we will introduce many other stereotypes to deal with reuse-contract-specific features.

3.3 Dependency relationships

Since the generalisation relationship of UML is insufficient to deal with all possible variants of incremental modification, we need to find another alternative. The *dependency relationship* is a common mechanism that can be applied to all modelling elements. It represents a unidirectional relationship from the source element to the target element, and indicates that the source *depends on* the target. It can be used to indicate situations in which changes to the target element may require changes to the source element. The dependency relationship is depicted by a dashed arrow from source to target, optionally labelled with a stereotype and a name. [Rational97] suggests attaching the «refine» stereotype to dependency relationships to represent a historical or derivation connection between two elements. All of this seems to indicate that the dependency relationship is a good candidate for representing incremental modification of modelling elements.

In the specific case of reuse contracts, the source of the dependency relationship corresponds to the provided component that is annotated with the «provider clause» stereotype. Likewise, the target element corresponds to the reused or evolved component, and is annotated with the «reuser clause» stereotype. The dependency arrow itself corresponds to an incremental modification of the provider (by reuse or evolution), and is annotated with the contract type that specifies in which way the provider is reused or evolved. For this reason, the «refine» stereotype of UML is specialised to specific stereotypes like «extension», «cancellation», «refinement» and «coarsening». There is a one-to-one correspondence between these stereotypes and the contract types. An example of an incremental modification using this notation is presented in Figure 4.

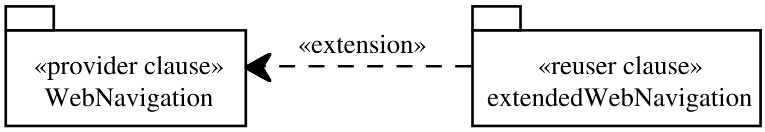

Figure 4: Incremental modification by extending the WebNavigation component

Attaching a specific contract type to the dependency relation 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. Formally, these obligations, permissions and prohibitions will need to be defined partly in the UML meta model, and partly in the natural language semantics of UML.

3.4 Design level diagrams

UML contains many different kinds of diagrams. To name but a few, we have use case diagrams, collaboration 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 design-level diagrams. More specifically, we will investigate class diagrams and collaboration diagrams. Together they can be used to describe the behaviour (at design level) of collaborating classes.

The static aspects of class collaborations can be represented in UML by means of *class diagrams*. They basically describe the entities that exist in the model (the classes), their internal structure (their attributes and operations), and their relationships with other entities. The dynamic aspects of the class collaboration can be represented by means of *interaction diagrams*. These describe the object interaction or message sending behaviour. In UML, two kinds of interaction diagrams are distinguished: *sequence diagrams* and c*ollaboration diagrams*.

We will use a *restricted* version of class diagrams to represent the static structure of the collaborating classes. For example, we only describe the class operations, not the class attributes. In other words, we only work with class interfaces (denoted with the «interface» stereotype). Moreover, classes can only be connected by means of binary associations. If a name is attached to the end of an association this indicates that the association is only navigable in this direction. If a name is attached to both ends of the association, the association is navigable in both directions. We do not allow associations to have adornments or classes to be connected by dependency or generalisation relationships. The incorporation of these features in our approach is regarded future work.

To represent the actual class collaboration, we will use a restricted form of collaboration diagrams. The main restriction is that we will not put an order on the message sends. Furthermore, we will not attach any stereotypes like «parameter», «local» or «global» to the associations. Adding these features is again considered future work.

While we have already seen in Figures 1 to 3 that the collaboration diagrams are encapsulated in a package annotated with the «interface» stereotype, the static structure diagrams will be encapsulated in a package annotated with the «static structure» stereotype. A static structure diagram together with a set of interaction diagrams 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.

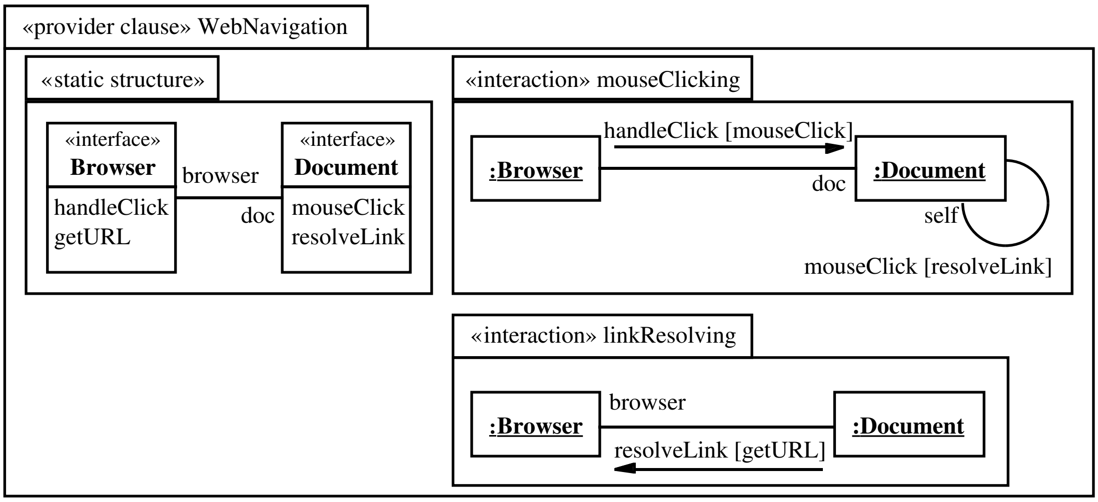

Figure 5: Design level diagrams for class collaboration

In Figure 5, an example of such a class collaboration is shown, namely the web navigation that was already explained partly in Figure 1. In the static structure we can see which operations and which classes are important from a design point of view, and what the associations are between the collaborating classes. Two interactions are used to express the essential behaviour: `linkResolving` and `mouseClicking`. The latter specifies what happens when the `Browser` receives a `handleClick` event.

4. Formalising Evolution of UML Diagrams

In this section we try to formally define those concepts that are needed to cope with evolution of modelling elements.

4.1 Formal definitions

First we need to formally define what our static structure diagrams and interaction diagrams look like. In UML, only an informal language definition of both kinds of diagrams is given.

Definition:

A **static structure** (P,A) is a couple consisting of a set P of participants and a set A of directed 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 **directed 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 directed 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 directed association we write $name(a)=n$, $source(a)=s$ and $target(a)=t$.

Definition:

An **interaction** (n,D) **associated to a static structure** S is a couple consisting of a name n and a set D of operation dependencies.

An **operation dependency** $d \in D$ 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 $dependencies(I)=D$.

If $d=(a,o,p)$ is an operation dependency we write $association(d)=a$, $caller(d)=o$, $callee(d)=p$, $source(d) = source(a)$ and $target(d) = target(a)$.

Note that the definition above always associates a static structure to an interaction! This is not the case in UML, that doesn't formally specify how the different kinds of diagrams are related to each other. Unlike static structures, interactions need to be distinguished by their name, since many different interactions can be associated to the same static structure, as we have already seen in Figure 5.

4.2 Maintaining the consistency of provider clauses

In Figure 5 we showed what our provider clauses look like. They usually contain a static structure diagram and a set of interaction diagrams. Obviously, since provider clauses describe reusable components, the interactions in a provider clause must be **consistent** with the static structure: the interactions can only contain instances of classes that are present in the static structure; all associations between objects in the interaction diagrams must be present in the static structure as well; and operation invocations from one object to another must satisfy the constraints that the operation signatures are present in the corresponding classes in the static structure. Since UML does not provide any notion of consistency between diagrams, we need to provide a definition ourselves.

Definition:

An interaction I associated to static structure S is **consistent with** S if

$\forall d \in dependencies(I)$: $caller(d) \in operations(source(d))$, $callee(d) \in operations(target(d))$

Consistency between interaction diagrams and the static structure alone is not enough. The different interactions in a provider clause need to be consistent with each other as well. For the moment, we have taken the easy solution where all interactions describe independent behaviour. This can be guaranteed by prohibiting each operation to play 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 $name(I_1) \neq name(I_2)$ and $\forall d_1 \in dependencies(I_1): \forall d_2 \in dependencies(I_2): caller(d_1) \neq caller(d_2)$

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 $d \in dependencies(I)$, with $I \in interactions(C)$, then we also write $d \in dependencies(C)$.

The provider clause given in Figure 5 satisfies the 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 static structure, since all the objects, associations and operations used in the interactions are defined in the static structure.

Although the provider clauses defined here were very simple, we already had to do a lot of work. 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. We are currently developing a tool that deals with these consistency issues between different kinds of UML diagrams.

4.3 Reuse contracts and contract types

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

Definition: A **reuser clause** C is a contract clause (that is not necessarily consistent).

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** (C_P,C_R,T) is a triple consisting of a provider clause C_P, a reuser clause C_R, and a contract type T.

The **contract type** T is an annotation that describes the relationship between the provider clause and the reuser clause.

Notation:

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

With these reuse contracts it is possible to detect evolution conflicts. The kind of conflicts that will occur depends 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* modifiers and *context* modifiers. 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 static structure diagram and describes participants and directed associations. The participant level can be found partially in the static structure and partially in the interaction diagrams and includes the operation signatures and the operation dependencies.

Basic Contract Type	Meaning
«participant extension»	adding new operation signatures to participants
«participant cancellation»	removing operation signatures from participants
«participant refinement»	adding new operation dependencies
«participant coarsening»	removing operation dependencies
«context extension»	adding new participants to the context
«context cancellation»	removing participants from the context
«context refinement»	adding new directed associations to the context
«context coarsening»	removing directed 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, operations and participants. *Refinement* introduces new dependencies between these elementary modelling elements: «participant refinement» connects operations by adding operation dependencies, «context refinement» connects participants by adding directed associations. *Cancellation* and *coarsening* are the inverses of extension and refinement, respectively. Together these operators 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 are added to the provider clause, extra contract types will be required.

Due to the poor support for incremental modification mechanisms in UML, or in any other object-oriented analysis and design notation, we need to define all these contract types ourselves. Each contract type is fully characterised by three definitions. The first one is the "*reuser clause definition*". This definition describes the form of a reuser clause modelling a particular contract type. Depending on the contract type the reuser clause will need to carry different kinds of information. The second definition needed is the "*applicability definition*". This definition describes what properties a reuser clause must comply with in order to be applicable to a particular provider clause. While the reuser clause definition is completely independent of possible provider clauses, the applicability definition specifies the relationship between a particular reuser clause and a particular provider clause. The third and last definition is the "*application definition*". It describes how the result of combining a provider and reuser clause for a particular contract type is determined. For each contract type the property holds that the result is again a provider clause.

For a full definition of all basic contract types we refer to [Lucas97][2]. We will only give the formal definition of one specific basic contract type as an example, namely «participant extension».

Reuser clause definition. For a participant extension, the only desired information 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, $\forall p \in participants(C): operations(p) \neq \emptyset$.

Applicability definition. For a 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 $operations(p) \cap operations(q)=\emptyset$.

Application definition. The result of a participant extension is obtained by augmenting the participants in the provider clause with the operation signatures in the reuser clause.

Definition:

C_{pe} is the **result of applying** a reuser clause C_R to a provider clause C_P **with respect to the type «participant extension»** if:

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

(2) $interactions(C_{pe}) = interactions(C_P)$

(3) $associations(C_{pe}) = associations(C_P)$

(4) $\forall q \in participants(C_P): \exists p \in participant(C_{pe})$ 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 $operation(p)=operations(q)$

Property: C_{pe} is a provider clause.

---

5. Detecting Evolution 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) evolution conflicts to be detected automatically.

5.1 Kinds of Evolution Conflicts

Disciplined reuse aims at maintaining a maximum degree of consistency between reusable components and the systems in which they are reused. In order to achieve this,

---

[2] Note that in [Lucas97] a different terminology was used. For example, what was called a *reuse operator* there is called a *contract type* here. Likewise, an *acquaintance relationship* there is called a *directed association* here.

both reuse and evolution of reusable components need to be documented by reuse contracts. This enables providing feedback on possible conflicts when composing or evolving components.

In general, conflicts can occur when two independent changes are made to the same model, regardless of whether this is achieved through composition, during evolution or by different developers. We therefore just consider two independent modifications of the same model and we investigate how these modifications can be reused together. This section inventorises a number of problems 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 a dependency to 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 provider clause needs to be consulted or the resulting provider clause needs to be computed. This is the case for conflicts that involve the transitive closure of dependencies. 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 dependencies on operations are removed. This can only be achieved by a «participant coarsening». For the conflict to occur, the second reuser clause should change the operation from which the dependency was removed, so it can only be a «participant refinement» or a «participant coarsening». This leads to the following rule.

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

(2) $\exists\, d_2 \in dependencies(C_2)$ 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 dependency 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, reconsider the `WebNavigation` provider clause of Figure 5. To obtain a `HistoryNavigation` web browser with history functionality, the `WebNavigation` provider clause is incrementally modified by first extending the `Browser` participant with a new operation signature `addURL`, and then refining the `Browser` participant by adding 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 6). 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.

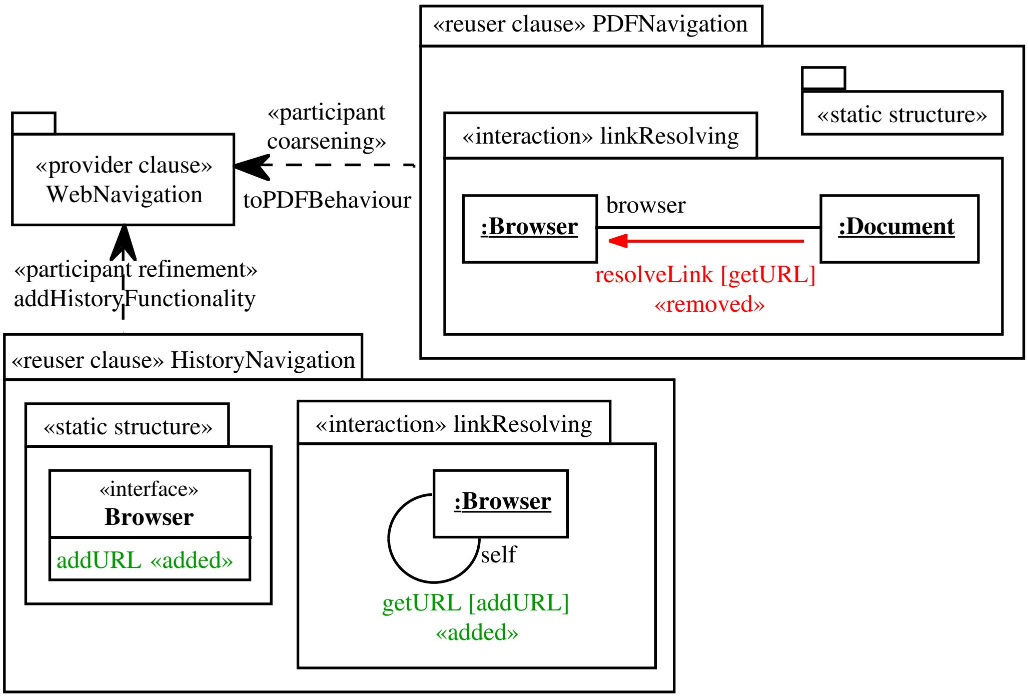

Figure 6: Detecting an inconsistent operations conflict

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 diagrams 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 that express different information. A second way is by nesting contract clauses inside each other (making use of the fact that packages can be nested), so that we can deal with reuse and evolution 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*. Again, the ability to nest packages allows us to make combined contract types arbitrarily complex.

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 evolution 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 evolution 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 scaleable 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 operation dependency on *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» as well as a «participant refinement», 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 dependency is removed during a «participant factorisation», this dependency is 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 tool support for reusable specifications and designs. In the case of UML, we claim that this is partly due to the lack of a precise semantics. Despite the powerful built-in extendibility mechanism of UML and the interesting packaging feature, its informal natural language semantics gives rise to a lot of problems when trying to incorporate reuse and evolution features.

In this paper, we have illustrated how the reuse contracts formalism can be incorporated into UML. In this way, a precise semantics is given to reuse and evolution of UML diagrams. More specifically, we have used packages to encapsulate reusable components, and dependency relationships between packages to express the actual reuse and evolution of these components.

We have shown that this precise semantics for reuse and evolution actually gives an added value to UML, since evolution 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. It is obvious that all these benefits will help in improving the software development process.

Although the results presented in this paper are too primitive to be practically useful, the ability to nest packages allows us to scale up the ideas, so that reuse contracts can be applied in real-life situations as well.

7. Future Work

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

While UML contains two important extension mechanisms, namely stereotypes and constraints, we have only used stereotypes to express evolution of reusable components. In some situations however, constraints might be useful as well. For example, for expressing additional constraints that should be fulfilled before the evolution can take place. How and where constraints should be used is a first topic of future work.

As a second topic, the proposed model should be enhanced further to support the full functionality of class diagrams and collaboration diagrams. A challenging task will be to incorporate order on operation dependencies in the collaboration diagrams. The main problem here is that it is still unclear how a collaboration diagram can be incrementally modified in a meaningful way. In the UML semantics, no details are given about generalisation of collaboration diagrams.

In order to be a full extension of UML, reuse contracts also need to be incorporated in the other analysis and design level diagrams like use case diagrams, sequence diagrams and state-transition diagrams. 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 now 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 change is made in one diagram, what is the impact on other diagrams? Related to this issue is the problem of transition between software phases. Until now, we have only described the dependencies between different diagrams in the same phase, namely the design phase. Another challenge is to try to describe the dependencies between diagrams in different phases of the software life-cycle by means of reuse contracts and contract types. This should lead to a better tractability between the software phases, and should enable assessing the impact of making changes to higher level diagrams on the associated lower level diagrams, but also about how modifying lower level diagrams makes them drift away from diagrams at the higher level. As a first step in this direction, we are planning to integrate use case diagrams in our model, and see how they interact with the design-level diagrams, more specifically, class diagrams and collaboration diagrams.

References

[Lucas97] Carine Lucas: "Documenting Reuse and Evolution with Reuse Contracts", PhD Dissertation, 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", 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.